Global Call IP for Host Media Processing

Technology Guide

May 2005

05-2239-005

Intel may make changes to specifications and product descriptions at any time, without notice.

This Global Call IP for Host Media Processing Technology Guide as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Publication Date: May 2005

Document Number: 05-2239-005

Intel Converged Communications, Inc.
1515 Route 10
Parsippany, NJ 07054

For **Technical Support**, visit the Intel Telecom Support Resources website at:
http://developer.intel.com/design/telecom/support

For **Products and Services Information**, visit the Intel Telecom Products website at:
http://www.intel.com/design/network/products/telecom

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products page at:
http://www.intel.com/buy/wtb/wtb1028.htm

Contents

Figures

Tables

Revision History

This revision history summarizes the changes made in each published version of this document.

Document No.	Publication Date	Description of Revisions
05-2039-005	April 2005	Call Control Library Initialization section: Added more detail about how to set configuration items before calling gc_Start( ) Setting and Retrieving SIP Message Header Fields section: Rewritten to document generic access mechanism and long header support Sending OPTIONS Requests section: added note on inclusion of MIME SDP body Responding to OPTIONS Requests section: Added information about automatic inclusion of SDP in MIME body of OK responses Sending NOTIFY Requests section: Corrected code example Using H.323 Annex M Tunneled Signaling Messages: New section and subsections Registration section: Reorganized subsections and added information on new SIP registration capabilities SIP Digest Authentication: New section and subsections Global Call Line Devices for Call Transfer section: Added requirements for endpoints being on same virtual board Debugging Global Call IP Applications chapter: Added note in two locations about SIP stack parsing error log file Global Call Functions Supported by IP section: Added entries for gc_SetAuthenticationInfo( ) and new gc_util_... functions IP-Specific Global Call Functions: New section to contain API reference pages for: gc_SetAuthenticationInfo( ) (new function), gc_util_copy_parm_blk( ) (new function), gc_util_find_parm_ex( ) (new function), gc_util_insert_parm_ref_ex( ) (new function), gc_util_next_parm_ex( ) (new function), INIT_IP_VIRTBOARD( ) INIT_IPCCLIB_START_DATA( ) Valid Extension IDs for the gc_Extension( ) Function table: Added note on parameter order requirement when using IPEXTID_SENDMSG gc_OpenEx( ) Variances for IP section: Added note about not closing and re-opening channels (PTR# 32087) gc_Start( ) Variances for IP section: Added information about how to reference configuration data structure when calling function Initialization Functions section: Eliminated section by moving information to API reference pages in new IP-Specific Global Call Functions section Summary of Parameter Sets and Parameter Usage table: Added two authentication parameter IDs to IPSET_CONFIG. Added “deprecated” indication to all parameters in the IPSET_SIP_MSGINFO set except IPPARM_SIP_HDR. Added six new entries for IPSET_TUNNELEDSIGNALMSG parameter set. IPSET_CONFIG Parameter Set table: Added IPPARM_AUTHENTICATION_CONFIGURE and IPPARM_AUTHENTICATION_REMOVE parameters IPSET_SIP_MSGINFO section: Added note on deprecation of most parm IDs. Added note about using extended gc_util_... functions with IPPARM_SIP_HDR IPSET_SIP_MSGINFO Parameter Set table: Added “deprecated” indications to all parameters except IPPARM_SIP_HDR

Document No.	Publication Date	Description of Revisions
05-2039-005 (continued)		IPSET_REG_INFO Parameter Set table: Added IP_REG_QUERY_INFO value for IPPARM_OPERATION_REGISTER parameter. Added IPPARM_REG_AUTOREFRESH and IPPARM_REG_SERVICEID parameters. IPSET_TUNNELEDSIGNALMSG parameter set: New section GC_PARM_DATA_EXT data structure: New section IP_AUTHENTICATION data structure: New section IP_TUNNELPROTOCOL_ALTID data structure: New section IP_VIRTBOARD data structure: Added new h323_msginfo_mask value for Annex M tunneled signaling messages. Added sip_registration_registrar field. IPCCLIB_START_DATA data structure: Added max_parm_data_size field Failure Response Codes When Using SIP section: Added subsection for new SIP Registration Error response codes
05-2039-004	January 2005	H.450.2 Blind Call Transfer Failure - Party B Rejects Call Transfer figure: Missing portion of figure restored Endpoint Behavior in H.450.2 Supervised Call Transfer section: Added precondition information, including parties in consultation call being in connected state Call Transfer Scenarios When Using SIP: New section and subsections Setting a SIP Outbound Proxy: New section Configuring SIP Transport Protocol: New section and subsections Retrieving Current Call-Related Information section: Added note about acknowledging call before extracting information in H.323 Standard Call-Related SIP Message Header Fields table: Added entries for ten additional headers Standard Call-Related Headers for Outbound SIP Messages: New table showing relationship between header s, Global Call functions, and SIPmessage types Standard Call-Related Headers for Inbound SIP Messages: ANew table showing relationship between SIP message types, Global Call event types, and headers Setting Additional (Generic) SIP Message Headers: New section Retrieving Additional (Generic) SIP Message Headers: New section Using SIP Messages with MIME Bodies (SIP-T): New section and subsections Global Call Events for Incoming SIP Messages that can Contain MIME Bodies table: Added five additional message types. Event for 3xx to 6xx responses changed from GCEV_TASKFAIL to GCEV_DISCONNECTED. Global Call Functions for SIP MIME Messages Using IPSET_MIME table: Added one function and five additional message types Specifying Transport for SIP Messages: new section Handling SIP Transport Failures: new section Sending and Receiving SIP INFO Messages: New section and subsections Sending and Receiving SIP OPTIONS Messages: New section and subsections Using SIP SUBSCRIBE and NOTIFY Messages: New section and subsections Specifying DTMF Support section: Clarified descriptions of bitmask values Getting Media Streaming Status and Connection Information section: Added information on getting local and remote RTP addresses Call Transfer When Using SIP: New section and subsection Sending a T.38 Fax in a Session Without Audio Established section: Corrected code example (PTR#33979)

Document No.	Publication Date	Description of Revisions
05-2039-004 (continued)		Receiving a T.38 Fax in a Session Without Audio Established section: Corrected code example (PTR#34073) Host LAN Disconnection Alarms: New section and subsection Debugging Global Call IP Applications chapter: Completely rewritten to describe new RTF logging facillities gc_AcceptInitXfer( ) Variances for IP section: Added SIP variances gc_AcceptXfer( ) Variances for IP section: Added SIP variances gc_Extension( ) Variances for IP section: Added IPEXTID_MSGINFO entry and added SIP message type in entries for IPEXTID_RECEIVEMSG and IPEXTID_SENDMSG in Valid Extension IDs for the gc_Extension( ) Function table gc_GetCallInfo( ) Variances for IP section: Added info on SIP-specificforms of origination address and destination address gc_InitXfer( ) Variances for IP section: Added SIP variances gc_InvokeXfer( ) Variances for IP section: Added SIP variances gc_RejectInitXfer( ) Variances for IP section: Added SIP variances gc_RejectXfer( ) Variances for IP section: Added SIP variances gc_SetConfigData( ) Variances for IP: Added SIP variance about enabling call transfer invoke acknowledge events gc_Start( ) Variances for IP: Added bullet items with default value info for SIP MIME, SIP outbound proxy, SIP transport protocol, SIP request retry, and SIP OPTIONS access configuration items INIT_IP_VIRTBOARD( ) section: Added info on SIP MIME enable, SIP outbound proxy, SIP transport protocol, SIP request retry, and SIP OPTIONS access Summary of Parameter Sets and Parameter Usage table: Added IPPARM_REGISTER_SIP_HDR in IPSET_CONFIG set; IPSET_MIME and IPSET_MIME200OK_TO_BYE sets (5 parameter IDs); IPSET_MSG_SIP set (2 parameter IDs); IPSET_RTP_ADDRESS set (2 parameter IDs); 10 parameter IDs in IPSET_SIP_MSGINFO plus additional function and event information; IPSET_SIP_REQUEST_ERROR (2 parameter IDs) IPSET_CONFIG section: Added IPPARM_REGISTER_SIP_HDR IPSET_MIME and IPSET_MIME_200OK_TO_BYE: New section IPSET_MSG_SIP: New section IPSET_RTP_ADDRESS: New section IPSET_SIP_MSGINFO: Added 10 additional parameter IDs IPSET_SIP_REQUEST_ERROR: New section IP_ADDR structure description: Corrected structure name in text and typedef (was IPADDR) IP_VIRTBOARD structure description: Corrected data type of localIP field. Added SIP MIME enable mask value and fields for SIP outbound proxy, SIP transport protocol, SIP request retry, and SIP OPTIONS access enable. RTP_ADDR structure description: New section

Document No.	Publication Date	Description of Revisions
05-2039-003	September 2004	General Conditions for Call Transfers section: New section Using Fast Start and Slow Start Setup section: Added note about H.323 fast start when no coder is specified (PTR#33321) Summary of Call-Related Information that can be Set table: Added note that GC_SINGLECALL must be used for Call ID and SIP Message Information fields. Added entries for four additional SIP Message Information fields. Retrievable Call Information table: Revised datatype for H.323 Call ID and added info on SIP Call ID Examples of Retrieving Call-Related Information section: Added code examples for retrieving and parsing Call ID. Supported SIP Message Information Fields table: Added entries for Call ID, Diversion URI, Referred-By, and Replaces. Updated Contact URI entry to indicate setting is supported. Nonstandard Registration Message section: Corrected parameters and added code example gc_GetCallInfo( ) Variances for IP section: Added information on getting Call ID. Added SIP-specific address formats (To URI and From URI) gc_MakeCall( ) Variances for IP section: Added note about SIP timeout Configurable Call Parameters When Using H.323 table: Corrected value names for IPPARM_CONNECTIONMETHOD. Added entry for IPSET_CALLINFO/IPPARM_CALLID. Configurable Call Parameters When Using SIP table: Corrected value names for IPPARM_CONNECTIONMETHOD. Added entry for IPSET_CALLINFO/IPPARM_CALLID. gc_Start( ) Variances for IP section: Added information on default board instances and parameter values Summary of Parameter Sets and Parameter Usage table: Updated info for IPSET_CALLINFO/IPPARM_CALLID. Added entries in IPSET_SIP_MSGINFO section for IPPARM_CALLID_HDR, IPPARM_DIVERSION_URI, IPPARM_REFERRED_BY, and IPPARM_REPLACES. Added set/send info for IPSET_SIP_MSGINFO/IPPARM_CONTACT_URI. IPSET_CALLINFO Parameter Set table: Updated description of IPPARM_CALLID. Corrected value names for IPPARM_CONNECTIONMETHOD. IPSET_SIP_MSGINFO Parameter Set table: Added entries for IPPARM_CALLID_HDR, IPPARM_DIVERSION_URI, IPPARM_REFERRED_BY, and IPPARM_REPLACES Updated IPPARM_CONTACT_URI to indicate that setting is supported. Added length defines for all parameters. IP_VIRTBOARD structure description: Added default values to field descriptions
05-2039-002	April 2004	Summary of Call-Related Information that can be Set table: Added entries for Call ID, MediaWaitForConnect, and PresentationIndicator. Coders Supported for Host Media Processing (HMP) table: Corrected G.711 entries to indicate VAD must be disabled (PTR 32576). Added row for G.729a. Corrected frame size for G.729a+b. Added row for T.38. (PTR 32623) Setting Busy Reason Codes: New section. Example of Retrieving Call-Related Information section: Corrected both example programs

Document No.	Publication Date	Description of Revisions
05-2039-002 (continued)		Generating or Detecting DTMF Tones Using a Voice Resource: New section Setting QoS Threshold Values and Retrieving QoS Threshold Values: Corrected ParmSetID name in both code examples (PTR 32690) Registration section: Corrected code example for SIP registration; added table to map abstract registrar registration concepts to SIP REGISTER elements Gatekeeper Registration Failure: New section. Global Call Functions Supported by IP section: Added bullet to indicate support for gc_GetCTInfo( ) gc_GetCTInfo( ) Variances for IP section: New section gc_MakeCall( ) Variances for IP section: Clarified procedure for setting protocol to use on multi-protocol devices. Added information to Forming a Destination Address String section about specifying port address in TCP/IP destination addresses. gc_ReqService( ) Variances for IP section: Added SIP support for alias gc_SetUserInfo( ) Variances for IP section: Added note about not using this function to set protocol to use on multi-protocol devices. gc_Start( ) Variances for IP sectio: Added note regarding network adaptor enabling/disabling. Added information about initialization functions and overriding defaults when appropriate. Initialization Functions: New section Summary of Parameter Sets and Parameter Usage table: Added IPPARM_MEDIAWAITFORCONNECT, IPPARM_PRESENTATION_IND, and IPPARM_PROGRESS_IND parameters to IPSET_CALLINFO Added IPSET_H323_RESPONSE_CODE/IPPARM_BUSY_CAUSE parameter Updated IPSET_LOCAL_ALIAS set entries to add SIP support Added IPSET_SIP_RESPONSE_CODE/IPPARM_BUSY_REASON parameter Parameter Set Reference section: Added and updated data type and size information for all parameter sets in section IPSET_CALLINFO section: Added entries for 3 new parameters IPSET_H323_RESPONSE_CODE: New section IPSET_REG_INFO section: Added row for IPPARM_REG_TYPE IPSET_SIP_MSGINFO section: Added section for parameters used when setting and retrieving SIP Message Information fields IPSET_SIP_RESPONSE_CODE: New section IP_VIRTBOARD structure description: Updated to refer to INIT_IP_VIRTBOARD() initialization function. Added sup_serv_mask, h323_msginfo_mask, and terminal_type fields (PTR 30491) IPADDR structure description: Added note that only supported ipv4 field value is IP_CFG_DEFAULT. Added info about byte order for IPv4 addresses. IPCCLIB_START_DATA structure description: Updated to refer to INIT_ IPCCLIB_START_DATA() initialization function. IP-Specific Event Cause Codes chapter: Updated descriptions of the possible event causes (PTR 31213:

Document No.	Publication Date	Description of Revisions
05-2039-001	September 2003	Initial production version of document. Much of the information contained in this document was previously published in the *Global Call IP over Host-based Stack Technology User's Guide*, document number 05-1512-005. Major changes as compared to 05-1512-005 include: Added T.38 Fax Server support Added Call Transfer support when using H.450.2 Added information about accessing SIP message information fields Specific changes include: Call Transfer Glare Condition: Added section and scenario diagram Specifying DTMF Support: Changed description of how to use GC_PARM_BLK to discover which DTMF modes are supported. Added information to IP-Specific Function Information for call transfer functions: gc_AcceptInitXfer( ), gc_AcceptXfer( ), gc_InitXfer( ), gc_InvokeXfer( ), gc_RejectInitXfer( ), gc_RejectXfer( ) gc_Start( ) Variances for IP: Added note on IP_CFG_MAX_AVAILABLE_CALLS

About This Publication

The following topics provide information about this publication.

- Purpose
- Intended Audience
- How to Use This Publication
- Related Information

Purpose

This guide is for users of the Global Call API writing applications that use host-based IP H.323 or SIP technology. The Global Call API provides call control capability and supports IP Media control capability. This guide provides Global Call IP-specific information only and should be used in conjunction with the *Global Call API Programming Guide* and the *Global Call API Library Reference*, which describe the generic behavior of the Global Call API.

This publication specifically documents the Global Call API for IP technology as it is implemented in the Intel® NetStructure™ Host Media Processing Software 1.3 for Windows* release. The Global Call API implementation in Intel® Dialogic® System Release software is documented in a separate set of documents.

Intended Audience

This guide is intended for:

- System Integrators
- Independent Software Vendors (ISVs)
- Value Added Resellers (VARs)
- Original Equipment Manufacturers (OEMs)

This publication assumes that the audience is familiar with the Windows* or Linux* operating system (as appropriate) and has experience using the C programming language.

How to Use This Publication

This guide is divided into the following chapters:

- Chapter 1, “IP Overview”, gives a overview of VoIP technology and brief introductions to the H.323 and SIP standards for novice users.

- Chapter 2, "Global Call Architecture for IP", describes how Global Call can be used with IP technology and provides an overview of the architecture.
- Chapter 3, "IP Call Scenarios" , provides some call scenarios that are specific to IP technology, including scenarios for the call transfer supplementary service.
- Chapter 4, "IP-Specific Operations", describes how to use Global Call to perform IP-specific operations, such as setting call related information, registering with a registration server, sending and receiving protocol-specific messages, etc.
- Chapter 5, "Building Global Call IP Applications" provides guidelines for building Global Call applications that use IP technology.
- Chapter 6, "Debugging Global Call IP Applications" provides information for debugging Global Call IP applications using RTF logging facilities.
- Chapter 7, "IP-Specific Function Information", documents functions that are specific to the IP technology and describes additional functionality or limitations for specific Global Call functions when used with IP technology.
- Chapter 8, "IP-Specific Parameters" provides a reference for IP-specific parameter set IDs and their associated parameter IDs.
- Chapter 9, "IP-Specific Data Structures", provides reference information for data structures that are specific to the use of Global Call with the IP technology.
- Chapter 10, "IP-Specific Event Cause Codes" describes IP-specific event cause codes.
- Chapter 11, "Supplementary Reference Information" provides supplementary information including technology references and formats for called and calling party addresses for H.323.
- A Glossary and an Index can be found at the end of the document.

Related Information

Refer to the following documents and web sites for more information about developing IP telephony applications that use the Global Call API:

- *Global Call API Programming Guide*
- *Global Call API Library Reference*
- *IP Media Library API Programming Guide*
- *IP Media Library API Library Reference*
- ITU-T Recommendation H.225.0, Call signalling protocols and media stream packetization for packet-based multimedia communication systems, *http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-H.225.0*
- ITU-T Recommendation H.245, Control protocol for multimedia communication, *http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-H.245*
- ITU-T Recommendation H.323, Packet-based multimedia communications systems, *http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-H.323*
- ITU-T Recommendation H.450.2, Call transfer supplementary service for H.323, *http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-H.450.2*
- Internet Engineering Task Force (IETF) Request for Comments RFC 1889, *RTP: A Transport Protocol for Real-Time Applications*, *http://ietf.org/rfc/rfc1889.txt*

- Internet Engineering Task Force (IETF) Request for Comments RFC 2976, *The SIP INFO Method*, *http://ietf.org/rfc/rfc2976.txt*
- Internet Engineering Task Force (IETF) Request for Comments RFC 3261, *SIP: Session Initiation Protocol*, *http://ietf.org/rfc/rfc3261.txt*
- Internet Engineering Task Force (IETF) Request for Comments RFC 3265, *Session Initiation Protocol (SIP)-Specific Event Notification* [SUBSCRIBE and NOTIFY methods], *http://ietf.org/rfc/rfc3265.txt*
- Internet Engineering Task Force (IETF) Request for Comments RFC 3515, *The Session Initiation Protocol (SIP) Refer Method*, *http://ietf.org/rfc/rfc3515.txt*
- *http://developer.intel.com/design/telecom/support* (for technical support)
- *http://www.intel.com/design/network/products/telecom* (for product information)

IP Overview 1

This chapter provides overview information about the following topics:

1.1 Introduction to VoIP

Voice over IP (VoIP) can be described as the ability to make telephone calls and send faxes over IP-based data networks with a suitable Quality of Service (QoS). The voice information is sent in digital form using discrete packets rather than via dedicated connections as in the circuit-switched Public Switch Telephone Network (PSTN).

At the time of writing this document, there are two major international groups defining standards for VoIP:

- International Telecommunications Union, Telecommunications Standardization Sector (ITU-T), which has defined the following:
  - Recommendation H.323, covering Packet-based Multimedia Communications Systems (including VoIP)
- Internet Engineering Task Force (IETF), which has defined drafts of the several RFC (Request for Comment) documents, including the following:
  - RFC 3261, the Session Initiation Protocol (SIP)

The H.323 recommendation was developed in the mid 1990s and is a mature protocol.

SIP (Session Initiation Protocol) is an emerging protocol for setting up telephony, conferencing, multimedia, and other types of communication sessions on the Internet.

1.2 H.323 Overview

The H.323 specification is an umbrella specification for the implementation of packet-based multimedia over IP networks that cannot guarantee Quality of Service (QoS). This section discusses the following topics about H.323:

- H.323 Entities
- H.323 Protocol Stack
- Codecs
- Basic H.323 Call Scenario

- Registration with a Gatekeeper
- H.323 Call Scenario via a Gateway

1.2.1 H.323 Entities

The H.323 specification defines the entity types in an H.323 network including:

Terminal
An endpoint on an IP network that supports the real-time, two-way communication with another H.323 entity. A terminal supports multimedia coders/decoders (codecs) and setup and control signaling.

Gateway
Provides the interface between a packet-based network (for example, an IP network) and a circuit-switched network (for example, the PSTN). A gateway translates communication procedures and formats between networks. It handles call setup and teardown and the compression and packetization of voice information.

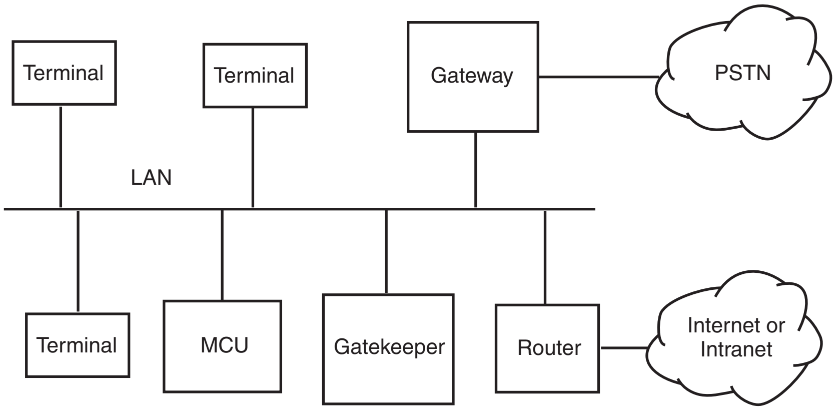
Gatekeeper
Manages a collection of H.323 entities in an H.323 zone controlling access to the network for H.323 terminals, Gateways, and MCUs and providing address translation. A zone can span a wide geographical area and include multiple networks connected by routers and switches. Typically there is only one gatekeeper per zone, but there may be an alternate gatekeeper for backup and load balancing. Typically, endpoints such as terminals, gateways, and other gatekeepers register with the gatekeeper.

Multipoint Control Unit (MCU)
An endpoint that supports conferences between three or more endpoints. An MCU can be a stand-alone unit or integrated into a terminal, gateway, or gatekeeper. An MCU consists of:

- Multipoint Controller (MC) – handles control and signaling for conferencing support
- Multipoint Processor (MP) – receives streams from endpoints, processes them, and returns them to the endpoints in the conference

Figure 1 shows the entities in a typical H.323 network.

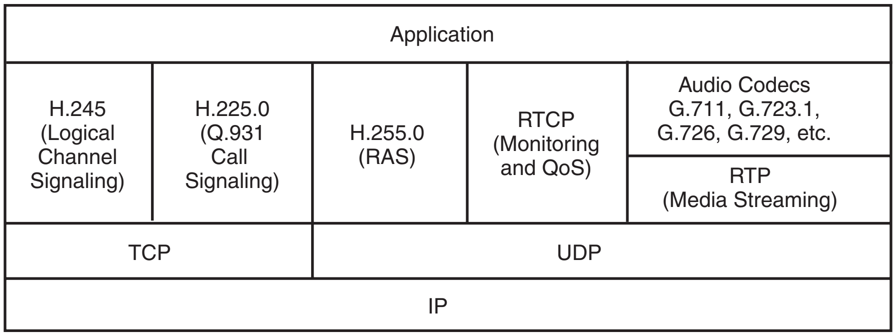
Figure 1. Typical H.323 Network



1.2.2 H.323 Protocol Stack

The H.323 specification is an umbrella specification for the many different protocols that comprise the overall H.323 protocol stack. Figure 2 shows the H.323 protocol stack.

Figure 2. H.323 Protocol Stack


Application
H.245 (Logical Channel Signaling)
H.225.0 (Q.931 Call Signaling)
H.255.0 (RAS)
RTCP (Monitoring and QoS)
Audio Codecs G.711, G.723.1, G.726, G.729, etc.
RTP (Media Streaming)
TCP
UDP
IP

The purpose of each protocol is summarized briefly as follows:

H.245
Specifies messages for opening and closing channels for media streams, and other commands, requests, and indications.

Q.931
Defines signaling for call setup and call teardown.

H.225.0
Specifies messages for call control, including signaling, the packetization and synchronization of media streams, and Registration, Admission, and Status (RAS).

Real Time Protocol (RTP)
The RTP specification is an IETF draft standard (RFC 1889) that defines the end-to-end transport of real-time data. RTP does not guarantee quality of service (QoS) on the transmission. However, it does provides some techniques to aid the transmission of isochronous data, including:

- information about the type of data being transmitted
- time stamps
- sequence numbers

Real Time Control Protocol (RTCP)
RTCP is part of the IETF RTP specification (RFC 1889) and defines the end-to-end monitoring of data delivery and QoS by providing information such as:

- jitter, that is, the variance in the delays introduced in transmitting data over a wire
- average packet loss

The H.245, Q.931, and H.225.0 combination provide the signaling for the establishment of a connection, the negotiation of the media format that will be transmitted over the connection, and call teardown at termination. As indicated in Figure 2, the call signaling part of the H.323 protocol is carried over TCP, since TCP guarantees the in-order delivery of packets to the application.

The RTP and RTCP combination is for media handling only. As indicated in Figure 2, the media part of the H.323 protocol is carried over UDP and therefore there is no guarantee that all packets will arrive at the destination and be placed in the correct order.

1.2.3 Codecs

RTP and RTCP data is the payload of a User Datagram Protocol (UDP) packet. Analog signals coming from an endpoint are converted into the payload of UDP packets by codecs (coders/decoders). The codecs perform compression and decompression on the media streams.

Different types of codecs provide varying sound quality. The bit rate of most narrow-band codecs is in the range 1.2 kbps to 64 kbps. The higher the bit rate the better the sound quality. Some of the most popular codecs are:

G.711
: Provides a bit rate of 64 kbps.

G.723.1
: Provides bit rates of either 5.3 or 6.4 kbps. Voice communication using this codec typically exhibits some form of degradation.

G.729
: Provides a bit rate of 8 kbps. This codec is very popular for voice over frame relay and for V.70 voice and data modems.

GSM
: Provides a bit rate of 13 kbps. This codec is based on a telephony standard defined by the European Telecommunications Standards Institute (ETSI). The 13 kbps bit rate is achieved with little degradation of voice-grade audio.

1.2.4 Basic H.323 Call Scenario

A simple H.323 call scenario can be described in five phases:

- Call Setup
- Capability Exchange
- Call Initiation
- Data Exchange
- Call Termination

Calls between two endpoints can be either direct or routed via a gatekeeper. This scenario describes a direct connection where each endpoint is a point of entry and exit of a media flow. The scenario described in this section assumes a slow start connection procedure. See Section 4.2, “Using Fast Start and Slow Start Setup”, on page 104 for more information on the difference between the slow start and fast start connection procedure.

The example in this section describes the procedure for placing a call between two endpoints, A and B, each with an IP address on the same subnet.

Call Setup

Establishing a call between two endpoints requires two TCP connections between the endpoints:

- One for the call setup (Q.931/H.225 messages)
- One for capability exchange and call control (H.245 messages)

Note: It is also possible to encapsulate H.245 media control messages within Q.931/H.225 signaling messages. The concept is known as *H.245 tunneling*. If tunneling is enabled, one less TCP port is required for incoming connections.

The caller at endpoint A connects to the callee at endpoint B on a well-known port, port 1720, and sends the call Setup message as defined in the H.225.0 specification. The Setup message includes:

- Message type; in this case, Setup
- Bearer capability, which indicates the type of call; for example, audio only
- Called party number and address
- Calling party number and address
- Protocol Data Unit (PDU), which includes an identifier that indicates which version of H.225.0 should be used along with other information

When endpoint B receives the Setup message, it responds with one of the following messages:

- Release Complete
- Alerting
- Connect
- Call Proceeding

In this case, endpoint B responds with the Alerting message. Endpoint A must receive the Alerting message before its setup timer expires. After sending this message, the user at endpoint B must either accept or refuse the call with a predefined time period. When the user at endpoint B picks up the call, a Connect message is sent to endpoint A and the next phase of the call scenario, Capability Exchange, can begin.

Capability Exchange

Call control and capability exchange messages, as defined in the H.245 standard, are sent on a second TCP connection. Endpoint A opens this connection on a dynamically allocated port at the endpoint B after receiving the address in one of the following H.225.0 messages:

- Alerting
- Call Proceeding
- Connect

This connection remains active for the entire duration of the call. The control channel is unique for each call between endpoints so that several different media streams can be present.

An H.245 TerminalCapabilitySet message that includes information about the codecs supported by that endpoint is sent from one endpoint to the other. Both endpoints send this message and wait for a reply which can be one of the following messages:

- TerminalCapabilitySetAck - accept the remote endpoints capability
- TerminalCapabilitySetReject - reject the remote endpoints capability

The two endpoints continue to exchange these messages until a capability set that is supported by both endpoints is agreed. When this occurs, the next phase of the call scenario, Call Initiation, can begin.

Call Initiation

Once the capability setup is agreed, endpoint A and B must set up the voice channels over which the voice data (media stream) will be exchanged. The scenario described here assumes a slow start connection procedure. See Section 4.2, "Using Fast Start and Slow Start Setup", on page 104 for more information on the difference between the slow start and fast start connection procedure.

To open a logical channel at endpoint B, endpoint A sends an H.245 OpenLogicalChannel message to endpoint B. This message specifies the type of data being sent, for example, the codec that will be used. For voice data, the message also includes the port number that endpoint B should use to send RTCP receiver reports. When endpoint B is ready to receive data, it sends an OpenLogicalChannelAck message to endpoint A. This message contains the port number on which endpoint A is to send RTP data and the port number on which endpoint A should send RTCP data.

Endpoint B repeats the process above to indicate which port endpoint A will receive RTP data and send RTCP reports to. Once these ports have been identified, the next phase of the call scenario, Data Exchange, can begin.

Data Exchange

Endpoint A and endpoint B exchange information in RTP packets that carry the voice data. Periodically, during this exchange both sides send RTCP packets, which are used to monitor the quality of the data exchange. If endpoint A or endpoint B determines that the expected rate of exchange is being degraded due to line problems, H.323 provides capabilities to make adjustments. Once the data exchange has been completed, the next phase of the call scenario, Call Termination, can begin.

Call Termination

To terminate an H.323 call, one of the endpoints, for example, endpoint A, hangs up. Endpoint A must send an H.245 CloseLogicalChannel message for each channel it has opened with endpoint B. Accordingly, endpoint B must reply to each of those messages with a CloseLogicalChannelAck message. When all the logical channels are closed, endpoint A sends an H.245 EndSessionCommand, waits until it receives the same message from endpoint B, then closes the channel.

Either endpoint (but typically the endpoint that initiates the termination) then sends an H.225.0 ReleaseComplete message over the call signalling channel, which closes that channel and ends the call.

1.2.5 Registration with a Gatekeeper

In a H.323 network, a gatekeeper is an entity that can manage all endpoints that can send or receive calls. Each gatekeeper controls a specific zone and endpoints must register with the gatekeeper to become part of the gatekeeper's zone. The gatekeeper provides call control services to the endpoints in its zone. The primary functions of the gatekeeper are:

- address resolution by translating endpoint aliases to transport addresses
- admission control for authorizing network access
- bandwidth management
- network management (in routed mode)

Endpoints communicate with a gatekeeper using the Registration, Admission, and Status (RAS) protocol. A RAS channel is an unreliable channel that is used to carry RAS messages (as described in the H.255 standard). The RAS protocol covers the following:

- Gatekeeper Discovery
- Endpoint Registration
- Endpoint Deregistration
- Endpoint Location
- Admission, Bandwidth Change and Disengage

Note: The RAS protocol covers status request, resource availability, nonstandard registration messages, unknown message response and request in progress that are not described in any detail in this overview. See *ITU-T Recommendation H.225.0 (09/99)* for more information.

Gatekeeper Discovery

An endpoint uses a process called *gatekeeper discovery* to find a gatekeeper with which it can register. To start this process, the endpoint can multicast a GRQ (gatekeeper request) message to the well-known discovery multicast address for gatekeepers. One or more gatekeepers may respond with a GCF (gatekeeper confirm) message indicating that it can act as a gatekeeper for the endpoint. If a gatekeeper does not want to accept the endpoint, it returns GRJ (gatekeeper reject). If more than one gatekeeper responds with a GCF message, the endpoint can choose which gatekeeper it wants to register with. In order to provide redundancy, a gatekeeper may specify an alternate gatekeeper in the event of a failure in the primary gatekeeper. Provision for the alternate gatekeeper information is provided in the GCF and RCF messages.

Endpoint Registration

An endpoint uses a process called *registration* to join the zone associated with a gatekeeper. In the registration process, the endpoint informs the gatekeeper of its transport, alias addresses, and endpoint type. Endpoints register with the gatekeeper identified in the gatekeeper discovery process described above. Registration can occur before any calls are made or periodically as necessary. An endpoint sends an RRQ (registration request) message to perform registration and in return receives an RCF (registration confirmation) or RRJ (registration reject) message.

Endpoint Deregistration

An endpoint may send an URQ (unregister request) in order to cancel registration. This enables an endpoint to change the alias address associated with its transport address or vice versa. The gatekeeper responds with an UCF (unregister confirm) or URJ (unregister reject) message.

The gatekeeper may also cancel an endpoint's registration by sending a URQ (unregister request) to the endpoint. The endpoint should respond with an UCF (unregister confirm) message. The endpoint should then try to re-register with a gatekeeper, perhaps a new gatekeeper, prior to initiating any calls.

Endpoint Location

An endpoint that has an alias address for another endpoint and would like to determine its contact information may issue a LRQ (location request) message. The LRQ message may be sent to a specific gatekeeper or multicast to the well-known discovery multicast address for gatekeepers. The gatekeeper to which the endpoint to be located is registered will respond with an LCF (location confirm) message. A gatekeeper that is not familiar with the requested endpoint will respond with LRJ (location reject).

Admission, Bandwidth Change and Disengage

The endpoint and gatekeeper exchange messages to provide admission control and bandwidth management functions. The ARQ (admission request) message specifies the requested call bandwidth. The gatekeeper may reduce the requested call bandwidth in the ACF (admission confirm) message. The ARQ message is also used for billing purposes, for example, a gatekeeper may respond with an ACF message just in case the endpoint has an account so the call can be charged. An endpoint or the gatekeeper may attempt to modify the call bandwidth during a call using a BRQ (bandwidth change request) message. An endpoint will send a DRQ (disengage request) message to the gatekeeper at the end of a call.

1.2.6 H.323 Call Scenario via a Gateway

While the call scenario described in Section 1.2.4, "Basic H.323 Call Scenario", on page 28 is useful for explaining the fundamentals of an H.323 call, it is not a realistic call scenario. Most significantly, the IP addresses of both endpoints were defined to be known in the example, while most Internet Service Providers (ISPs) allocate IP addresses to subscribers dynamically. This section describes the fundamentals of a more realistic example that involves a gateway.

A gateway provides a bridge between different technologies; for example, an H.323 gateway (or IP gateway) provides a bridge between an IP network and the PSTN. Figure 3 shows a configuration that uses a gateway. User A is at a terminal, while user B is by a phone connected to the PSTN.

Figure 3. Basic H.323 Network with a Gateway

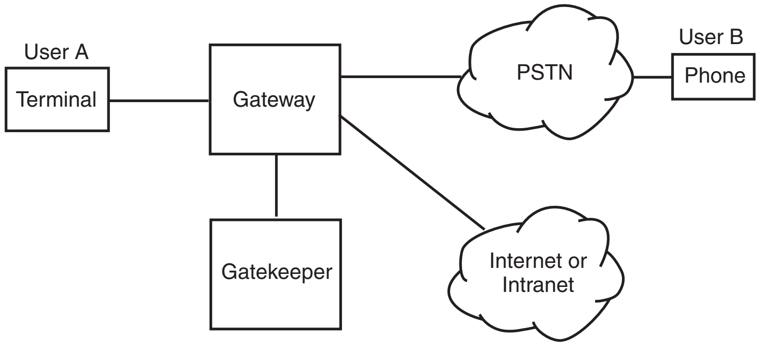


Figure 3 also shows a gatekeeper. The gatekeeper provides network services such as Registration, Admission, and Status (RAS) and address mapping. When a gatekeeper is present, all endpoints managed by the gatekeeper must register with the gatekeeper at startup. The gatekeeper tracks which endpoints are accepting calls. The gatekeeper can perform other functions also, such as redirecting calls. For example, if a user does not answer the phone, the gatekeeper may redirect the call to an answering machine.

The call scenario in this example involves the following phases:

- Establishing Contact with the Gatekeeper
- Requesting Permission to Call
- Call Signaling and Data Exchange
- Call Termination

Establishing Contact with the Gatekeeper

The user at endpoint A attempts to locate a gatekeeper by sending out a Gatekeeper Request (GRQ) message and waiting for a response. When it receives a Gatekeeper Confirm (GCF) message, the endpoint registers with the Gatekeeper by sending the Registration Request (RRQ) message and waiting for a Registration Confirm (RCF) message. If more than one gatekeeper responds, endpoint A chooses only one of the responding gatekeepers. The next phase of the call scenario, Requesting Permission to Call, can now begin.

Requesting Permission to Call

After registering with the gatekeeper, endpoint A must request permission from the gatekeeper to initiate the call. To do this, endpoint A sends an Admission Request (ARQ) message to the gatekeeper. This message includes information such as:

- a sequence number
- a gatekeeper assigned identifier
- the type of call; in this case, point-to-point
- the call model to use, either direct or gatekeeper-routed
- the destination address; in this case, the phone number of endpoint B

- an estimation of the amount of bandwidth required. This parameter can be adjusted later by a Bandwidth Request (BRQ) message to the gatekeeper.

If the gatekeeper allows the call to proceed, it sends an Admission Confirm (ACF) message to endpoint A. The ACF message includes the following information:

- the call model used
- the transport address and port to use for call signaling (in this example, the IP address of the gateway)
- the allowed bandwidth

All setup has now been completed and the next phase of the scenario, Call Signaling and Data Exchange, can begin.

Call Signaling and Data Exchange

Endpoint A can now send the Setup message to the gateway. Since the destination phone is connected to an analog line (the PSTN), the gateway goes off-hook and dials the phone number using dual tone multifrequency (DTMF) digits. The gateway therefore is converting the H.225.0 signaling into the signaling present on the PSTN. Depending on the location of the gateway, the number dialed may need to be converted. For example, if the gateway is located in Europe, then the international dial prefix will be removed.

As soon as the gateway is notified by the PSTN that the phone at endpoint B is ringing, it sends the H.225.0 Alerting message as a response to endpoint A. As soon as the phone is picked up at endpoint B, the H.225.0 Connect message is sent to endpoint A. As part of the Connect message, a transport address that allows endpoint A to negotiate codecs and media streams with endpoint B is sent.

The H.225.0 and H.245 signaling used to negotiate capability, initiate and call, and exchange data are the same as that described in the basic H.323 call scenario. See the Capability Exchange, Call Initiation, and Data Exchange phases in Section 1.2.4, "Basic H.323 Call Scenario", on page 28 for more information.

In this example the destination phone is analog, therefore, it requires the gateway to detect the ring, busy, and connect conditions so it can respond appropriately.

Call Termination

As in the basic H.323 call scenario example, the endpoint that hangs up first needs to close all the channels that were open using the H.245 CloseLogicalChannel message. If the gateway terminates first, it sends an H.245 EndSessionCommand message to endpoint A and waits for the same message from endpoint A. The gateway then closes the H.245 channel.

When all channels between endpoint A and the gateway are closed, each must send a DisengageRequest (DRQ) message to the gatekeeper. This message lets the gatekeeper know that the bandwidth is being released. The gatekeeper sends a DisengageConfirm (DCF) message to both endpoint A and the gateway.

1.3 SIP Overview

Session Initiation Protocol (SIP) is an ASCII-based, peer-to-peer protocol designed to provide telephony services over the Internet. The SIP standard was developed by the Internet Engineering Task Force (IETF) and is one of the most commonly used protocols for VoIP implementations. This section discusses the following topics about SIP:

- Advantages of Using SIP
- SIP User Agents and Servers
- Basic SIP Operation
- Basic SIP Call Scenario
- SIP Messages

1.3.1 Advantages of Using SIP

Some of the advantages of using SIP include:

- The SIP protocol stack is smaller and simpler than other commonly used VoIP protocols, such as H.323.
- SIP-based systems are more easily scalable because of the peer-to-peer architecture used. The hardware and software requirements for adding new users to SIP-based systems are greatly reduced.
- Functionality is distributed over different components. Control is decentralized. Changes made to a component have less of an impact on the rest of the system.
- SIP is Internet-enabled.

1.3.2 SIP User Agents and Servers

User agents (UAs) are appliances or applications, such as SIP phones, residential gateways and software that initiate and receive calls over a SIP network.

Servers are application programs that accept requests, service requests and return responses to those requests. Examples of the different types of servers are:

Location Server

Used by a SIP redirect or proxy server to obtain information about the location of the called party.

Proxy Server

An intermediate program that operates as a server and a client and which makes requests on behalf of the client. A proxy server does not initiate new requests, it interprets and possibly modifies a request message before forwarding it to the destination.

Redirect Server

Accepts a request from a client and maps the address to zero or more new addresses and returns the new addresses to the client. The server does not accept calls or generate SIP requests on behalf of clients.

Registrar Server

Accepts REGISTER requests from clients. Often, the registrar server is located on the same physical server as the proxy server or redirect server.

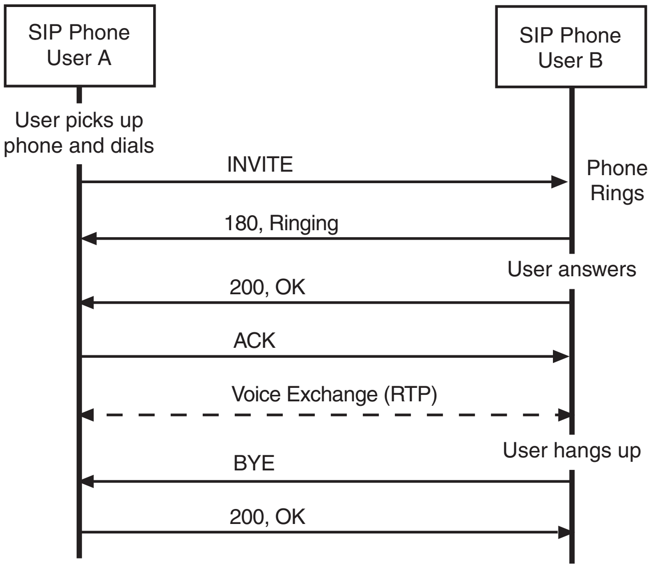
1.3.3 Basic SIP Operation

Callers and callees are identified by SIP addresses. When making a SIP call, a caller first locates the appropriate server and then sends a SIP request. The most common SIP operation is the invitation request. Instead of directly reaching the intended callee, a SIP request may be redirected or may trigger a chain of new SIP requests by proxies. Users can register their location(s) with SIP servers.

1.3.4 Basic SIP Call Scenario

Figure 4 shows the basic SIP call establishment and teardown scenario.

Figure 4. Basic SIP Call Scenario



1.3.5 SIP Messages

In SIP, there are two types of messages:

- SIP Request Messages
- SIP Response Messages

SIP Request Messages

The most commonly used SIP request messages are:

- INVITE
- ACK
- BYE
- REGISTER
- CANCEL
- OPTIONS

For more information, see RFC 3261 at *http://ietf.org/rfc/rfc3261.txt*.

SIP Response Messages

SIP response messages are numbered. The first digit in each response number indicates the type of response. The response types are as follows:

1xx
Information responses; for example, 180 Ringing

2xx
Successful responses; for example, 200 OK

3xx
Redirection responses; for example, 302 Moved Temporarily

4xx
Request failure responses; for example, 402, Forbidden

5xx
Server failure responses; for example, 504, Gateway Timeout

6xx
Global failure responses; for example, 600, Busy Everywhere

For more information, see RFC 3261 at the URL given above.

Global Call Architecture for IP 2

This chapter discusses the following topics:

2.1 Global Call over IP Architecture with a Host-Based Stack

Global Call provides a common call control interface that is independent of the underlying network interface technology. While Global Call is primarily concerned with call control, that is, call establishment and teardown, Global Call provides some additional capabilities to support applications that use IP technology.

Global Call support for IP technology includes:

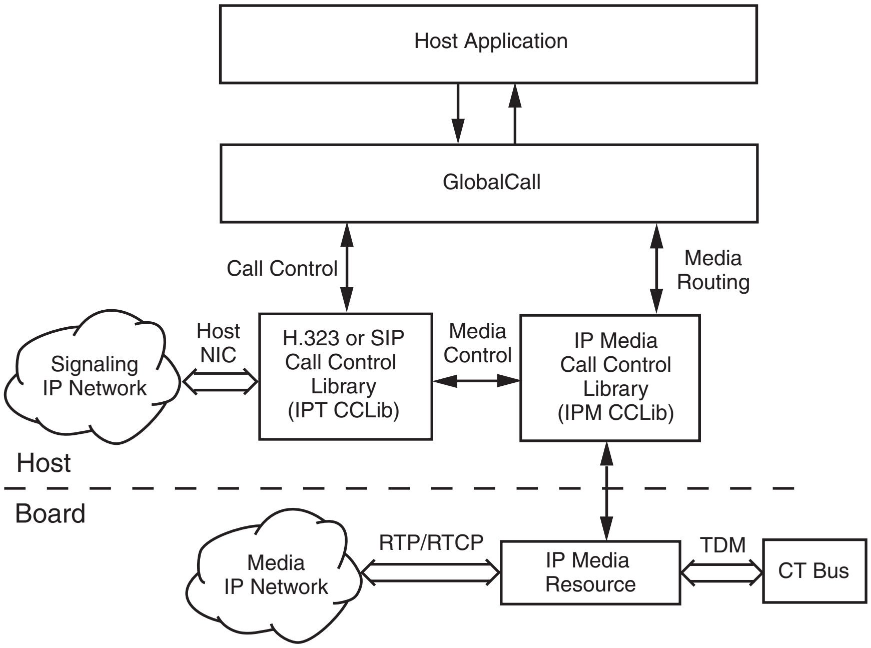
- call control capabilities for establishing calls over an IP network
- support for IP Media control by providing the ability to open and close IP Media channels for streaming

Global Call supports a system configuration where the IP signaling stack runs on the host and a Host Media Processing virtual board provides the IP resources for media processing.

Note: Global Call supports the RADVISION* H.323 and SIP stacks. If other third-party call control stacks are used, Global Call cannot be used for IP call control, but the IP Media Library can be used for media resource management. See the *IP Media Library API Programming Guide* and *IP Media Library API Library Reference* for more information.

Figure 5 shows the Global Call over IP architecture when using a DM3 board or an Intel NetStructure IPT board and a host-based stack provided with the system software

Figure 5. Global Call over IP Architecture Using a Host-Based Stack



To simplify IP Media management by the host application and to provide a consistent look and feel with other Global Call technology call control libraries, the IP Signaling call control library (IPT CCLib) controls the IP Media functionality.

2.2 Architecture Components

The role of each major component in the architecture is described in the following sections:

- Host Application
- Global Call
- IP Signaling Call Control Library (IPT CCLib)
- IP Media Call Control Library (IPM CCLib)
- IP Media Resource

2.2.1 Host Application

The host application manages and monitors the IP telephony system operations. Typically the application performs the following tasks:

- initializes Global Call
- opens and closes IP line devices
- opens and closes IP Media devices
- opens and closes PSTN devices

- configures IP Media and network devices (capability list, operation mode, etc.)
- performs call control, including making calls, accepting calls, answering calls, dropping calls, releasing calls, and processing call state events
- queries call and device information
- handles PSTN alarms and errors

2.2.2 Global Call

Global Call hides technology and protocol-specific information from the host application and acts as an intermediary between the host application and the technology call control libraries. It performs the following tasks:

- performs high-level call control using the underlying call control libraries
- maintains a generic call control state machine based on the function calls used by an application and call control library events
- collects and maintains data relating to resources
- collects and maintains alarm data

2.2.3 IP Signaling Call Control Library (IPT CCLib)

The IP Signaling call control library (IPT CCLib) implements IP technology. It performs the following tasks:

- controls the H.323 and/or SIP stack
- manages IP Media resources as required by the Global Call call state model and the IP signaling protocol model
- translates between the Global Call call model and IP signaling protocol model
- processes Global Call call control library interface commands
- generates call control library interface events

2.2.4 IP Media Call Control Library (IPM CCLib)

The IP Media Call Control Library (IPM CCLib) performs the following tasks:

- processes Global Call call control library interface commands for the opening, closing, and timeslot routing of media devices
- configures QoS thresholds
- translates QoS alarms to Global Call alarm events

2.2.5 IP Media Resource

The IP Media Resource processes the IP Media stream. It performs the following tasks:

- encodes PCM data from the TDM bus into IP packets sent to the IP network
- decodes IP packets received from the IP network into PCM data transmitted to the TDM bus
- configures and reports QoS information to the IP Media stream

2.3 Device Types and Usage

This section includes information about device types and usage:

- Device Types Used with IP
- IPT Board Devices
- IPT Network Devices
- IPT Start Parameters

2.3.1 Device Types Used with IP

When using Global Call with IP technology, a number of different device types are used:

IPT Board Device

A virtual entity that represents a NIC or NIC address (if one NIC supports more than one IP address). The format of the device name is **iptBx**, where **x** is the logical board number that corresponds to the NIC or NIC address. See Section 2.3.2, "IPT Board Devices", on page 43 for more information.

IPT Network Device

Represents a logical channel over which calls can be made. This device is used for call control (call setup and tear down). The format of the device name is **iptBxTy**, where **x** is the logical board number and **y** is the logical channel number. See Section 2.3.3, "IPT Network Devices", on page 44 for more information.

IP Media Device

Represents a media resource that is used to control RTP streaming, monitoring Quality of Service (QoS) and the sending and receiving of DTMF digits. The format of the device name is **ipmBxCy**, where **x** is the logical board number and **y** is the logical channel number.

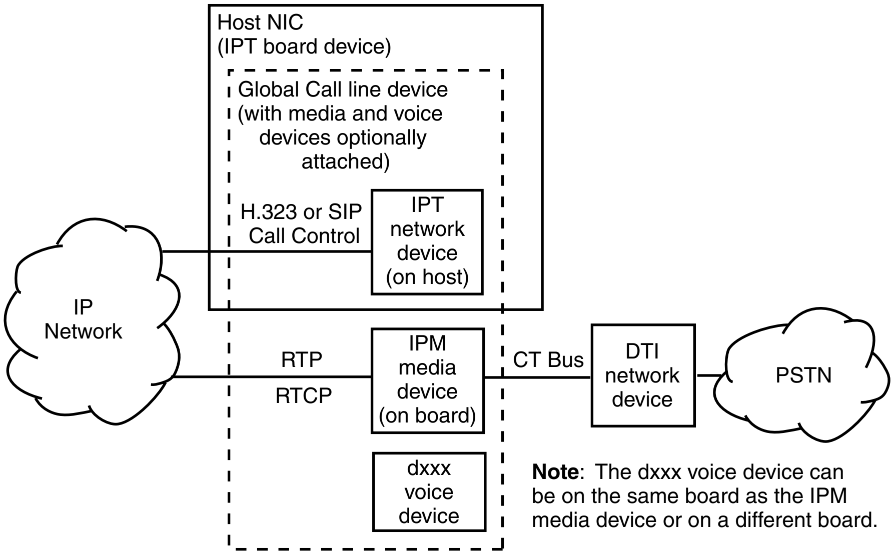
The IPT network device (iptBxTy) and the IP Media device (ipmBxCy) can be opened simultaneously in the same **gc_OpenEx( )** command. If a voice resource is available in the system, for example an IP board that provides voice resources or any other type of board that provides voice resources, a voice device can also be included in the same **gc_OpenEx( )** call to provide voice capabilities on the logical channel. See Section 7.3.17, "gc_OpenEx( ) Variances for IP", on page 326 for more information.

Alternatively, the IPT network device (iptBxTy) and the IP Media device (ipmBxCy) can be opened in separate **gc_OpenEx( )** calls and subsequently attached using the **gc_AttachResource( )** function.

The IP Media device handle, which is required for managing Quality of Service (QoS) alarms for example, can be retrieved using the **gc_GetResourceH( )** function. See Section 4.22, “Managing Quality of Service Alarms”, on page 214 for more information.

Figure 6 shows the relationship between the various types of Global Call devices when a single Host NIC is used.

Figure 6. Global Call Devices

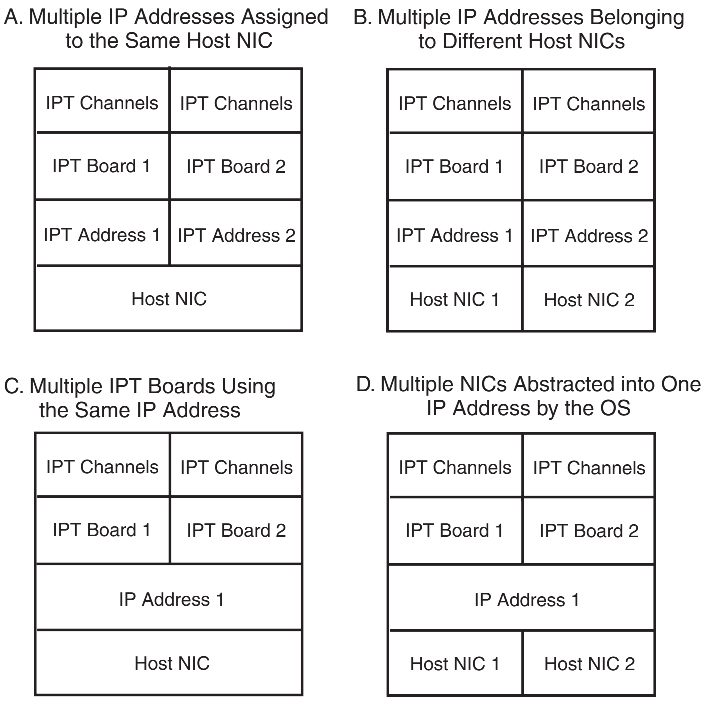


Note: The dxxx voice device can be on the same board as the IPM media device or on a different board.

2.3.2 IPT Board Devices

An IPT board device is a virtual entity that corresponds to an IP address and is capable of handling both H.323 and SIP protocols. The application uses the **gc_Start( )** function to bind IP addresses to IPT virtual board devices. Possible configurations are shown in Figure 7. The operating system must support the IP address and underlying layers before the Global Call application can take advantage of the configurations shown in Figure 7. Up to eight virtual IPT boards can be configured in one system. For each virtual IPT board, it is possible to configure the local address and signaling port (H.323 and SIP), the number of IPT network devices that can be opened simultaneously, etc. See Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340 for more information on how to configure IPT board devices.

Figure 7. Configurations for Binding IPT Boards to NIC IP Addresses



Note: IPT Board 1 and IPT Board 2 must have different port numbers.

Once the IPT board devices are configured, the application can open line devices with the appropriate IPT network device (IPT channel) and optionally IPT Media device (IPM channel).

The **gc_SetConfigData( )** function can be used on an IPT board device to apply parameters to all IPT channels associated with the IPT board device. The application can use the **gc_AttachResource( )** and **gc_Detach( )** functions to load balance which host NIC makes a call for a particular IPT media device (IPM channel). It is also possible that the operating system can perform load balancing using the appropriate NIC for call control as shown in Figure 7, configuration D.

The **gc_ReqService( )** function is used on an IPT board device for registration with an H.323 gatekeeper or SIP registrar. See Section 7.3.21, "gc_ReqService( ) Variances for IP", on page 329 for more information.

2.3.3 IPT Network Devices

Global Call supports three types of IPT network devices:

- H.323 only (P_H323 in the **devicename** string when opening the device)
- SIP only (P_SIP in the **devicename** string when opening the device)
- Dual protocol, H.323 and SIP (P_IP in the **devicename** string when opening the device)

The device type is determined when using the **gc_OpenEx( )** function to open the device. H.323 and SIP only devices are capable of initiating and receiving calls of the selected protocol type only.

Dual protocol devices are capable of initiating and receiving calls using either the H.323 or SIP protocol. The protocol used by a call on a dual protocol device is determined during call setup as follows:

- for outbound calls, by a parameter to the **gc_MakeCall( )** function
- for inbound calls, by calling **gc_GetCallInfo( )** to retrieve the protocol type used. In this case, the application can query the protocol type of the current call after the call is established, that is, as soon as either GCEV_DETECTED (if enabled) or GCEV_OFFERED is received.

2.3.4 IPT Start Parameters

The application determines the number of boards that will be created by the IPT call control library (up to the number of available IP addresses). For each board, the host application will provide the following information:

- number of line devices on the board
- maximum number of IPT devices to be used for H.323 calls (used for H.323 stack allocation)
- maximum number of IPT devices to be used for SIP calls (used for SIP stack allocation)
- board IP address
- signaling port for H.323
- signaling port for SIP
- enable/disable access to SIP message information fields (headers)
- enable/disable MIME-encoded SIP messages
- number and size of buffers in MIME memory pool (if MIME feature is enabled)
- enable/disable access to H.323 Message Information fields
- enable/disable call transfer supplementary service
- set terminal type
- enable and configure outbound proxy
- configure SIP transport protocol (enable use of TCP)
- configure SIP request retry behavior
- enable/disable application access to SIP OPTIONS messages

IP Call Scenarios 3

This chapter provides common call control scenarios when using Global Call with IP technology. Topics include:

3.1 Basic Call Control Scenarios When Using IP Technology

This section provides details of the basic call control scenarios when using IP technology. The scenarios include:

- Basic Call Setup When Using H.323 or SIP
- Basic Call Teardown When Using H.323 or SIP

3.1.1 Basic Call Setup When Using H.323 or SIP

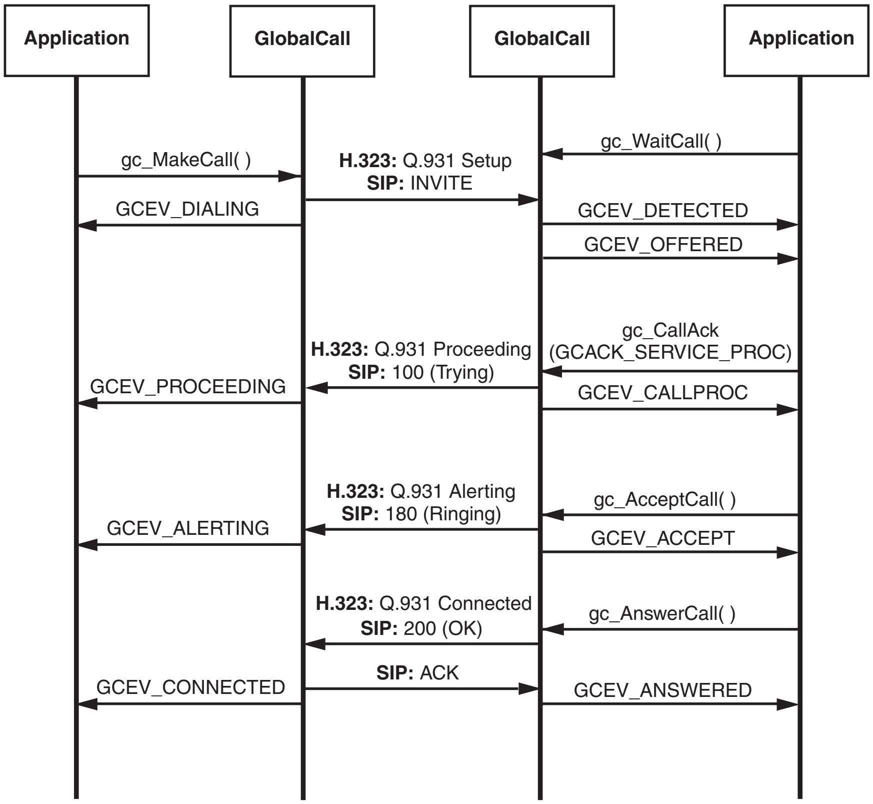
Figure 8 shows the basic call setup sequence when using H.323 or SIP.

Notes: **1.** This figure assumes that the network and media channels are already open and a media channel with the appropriate media capabilities is attached to the network channel. See Section 7.3.17, "gc_OpenEx( ) Variances for IP", on page 326 for information on opening and attaching network and media devices and Section 7.3.16, "gc_MakeCall( ) Variances for IP", on page 311 for detailed information on the specification of the destination address etc.

2. Only H.225.0 (Q.931) messages are shown in the sequence below. H.245 messages were omitted in the interest of simplification.

3. The destination address must be a valid address that can be translated by the remote node.

Figure 8. Basic Call Setup When Using H.323 or SIP

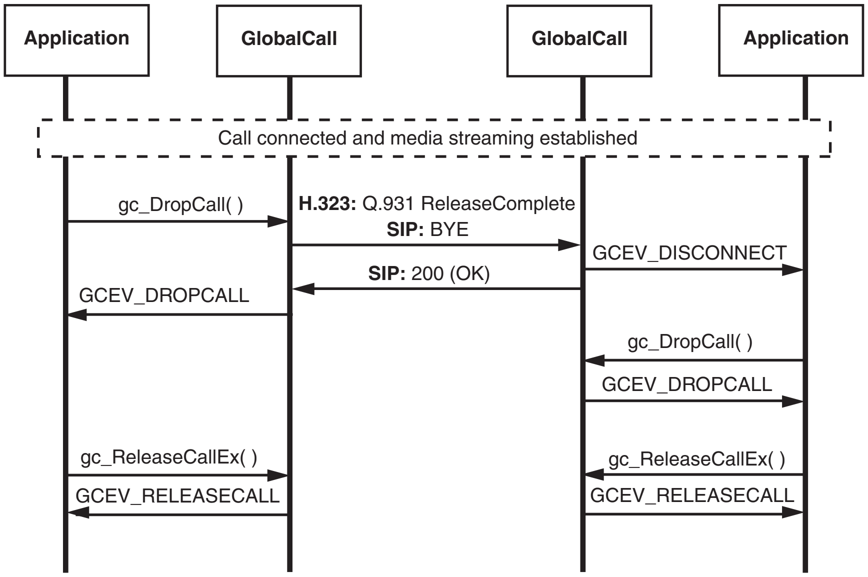


3.1.2 Basic Call Teardown When Using H.323 or SIP

Figure 9 shows the basic call teardown scenario when using Global Call with H.323 or SIP.

Note: Only H.225.0 (Q.931) messages are shown in the sequence below. H.245 messages were omitted in the interest of simplification.

Figure 9. Basic Call Teardown When Using H.323 or SIP



3.2 Call Transfer Scenarios When Using H.323

The Global Call API functions that supports IP call transfer are described in the *Global Call API Library Reference*. Information on implementing H.450.2 call transfer can be found in Section 4.25, “Call Transfer”, on page 237, and protocol-specific information about the individual call transfer APIs can be found in the subsections of Section 7.3, “Global Call Function Variances for IP”, on page 296.

The following topics describe the call transfer capabilities provided when using the H.450.2 supplementary service with H.323:

- General Conditions for H.450.2 Call Transfers
- Endpoint Behavior in H.450.2 Blind Call Transfers
- Successful H.450.2 Blind Call Transfer Scenario
- Unsuccessful H.450.2 Blind Call Transfer Scenarios
- Endpoint Behavior in H.450.2 Supervised Call Transfer
- Successful H.450.2 Supervised Call Transfer Scenario
- Unsuccessful H.450.2 Supervised Transfer Scenarios

3.2.1 General Conditions for H.450.2 Call Transfers

When performing a call transfer operation, all involved call handles must be on the same stack instance. This imposes the following application restrictions for call transfer operations:

- When performing a supervised call transfer at party A, both the consultation line device and the transferring line device must be on the same virtual board.
- When performing a call transfer (either supervised or blind) at party B, both the transferring line device and the transferred line device must be on the same virtual board.
- When performing a supervised call transfer at party C, both the consultation line device and the transferred-to line device must be on the same virtual board.

3.2.2 Endpoint Behavior in H.450.2 Blind Call Transfers

This section describes the behavior of each of the three endpoints in an H.450.2 blind call transfer. The assumed precondition for supervised call transfer is:

- The transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active call. From the perspective of the Global Call API, party A and party B are both in the GCST_CONNNECTED state.

3.2.2.1 Transferring Endpoint (Party A) Role

The transferring endpoint (party A) initiates the blind transfer by calling the **gc_InvokeTransfer( )** function, which results in the sending an ctInitiate.Invoke APDU in a FACILITY message. From this point forward, this endpoint is only subsequently notified as to the creation of the transferred call attempt. Note however, that it is not notified as to the end result of the transfer, specifically whether the transfer results in a connection or a no-answer. Instead, the transferring endpoint is only guaranteed notification that the transferred-to endpoint has been alerted to the incoming transferred call offering (that is, ringback). As specified in H.450.2, the ctInitiate.ReturnResult APDU may be returned in either ALERTING or CONNECT. The primary call will also be disconnected remotely via the transferred endpoint (party B) as part of a successful status notification from this endpoint. Both the forward and reverse logical channels will be closed along with their associated audio or data streams. From the Global Call API perspective, the primary call is terminated at the transferring endpoint, as indicated by the GCEV_DISCONNECTED event, implying the endpoint is then responsible for the drop and release of the primary call.

3.2.2.2 Transferred Endpoint (Party B) Role

The endpoint to be transferred (party B) is notified of the request to transfer from the initiating endpoint via the GCEV_REQ_XFER event. Assuming the party to be transferred accepts the transfer request via the **gc_AcceptXfer( )** function, it retrieves the destination address information from the unsolicited transfer request via the GC_REROUTING_INFO structure passed within the GCEV_REQ_XFER event. The endpoint to be transferred then uses the rerouting address information to initiate a call to the new destination party via **gc_MakeCall( )**. From the perspective of the application, this transferred call is treated in the same manner as a normal singular call and the party receives intermediate call state events as to the progress of the call (that is, GCEV_DIALING, GCEV_ALERTING, GCEV_PROCEEDING, and GCEV_CONNECTED). When the transferred endpoint receives its first indication from the transferred-to endpoint (party C) that the call transfer was successful (CTSetup.ReturnResult APDU), the transferred endpoint is notified of the transfer success and implicitly, without user or application initiation, disconnects the primary call with the transferring endpoint.

Assuming the transferred call is answered, the transferred endpoint is then involved in active media streaming with the transferred-to endpoint. Note that the notification of transfer success via the GCEV_XFERRED_CMPLT event may also arrive with any call progress events, that is, GCEV_ALERTING, GCEV_PROCEEDING, or GCEV_CONNECTED. Although the primary call to the transferring endpoint (party A) is implicitly dropped, the call itself must still be explicitly dropped via **gc_DropCall( )** to resynchronize the local state machine and released via **gc_ReleaseCallEx( )**.

3.2.2.3 Transferred-To Endpoint (Party C) Role

For the most part, from the perspective of the transferred-to endpoint (party C), the transferred call is treated as a typical incoming call. The call is first notified to the application via GCEV_DETECTED or GCEV_OFFERED events at which point the GCRV_XFERCALL cause value provided in the event will alert the application that this call offering is the result of a transfer. At that point, the application may retrieve the typical calling party information about the call. The transferred-to party is then provided the same methods of action as a typical incoming call, namely alerting the transferred endpoint (party B) that the call is proceeding (typical for gateways), ringback notification that the local user is being alerted, or simply answering the call. The only behavior change from this endpoint over typical non-transferred calls, is whether to treat or display the calling party information any differently if it is the result of a transfer. Assuming the transferred call is eventually connected or timed out on no answer, the transferred-to party must eventually drop and release this call as the case for non-transferred call.

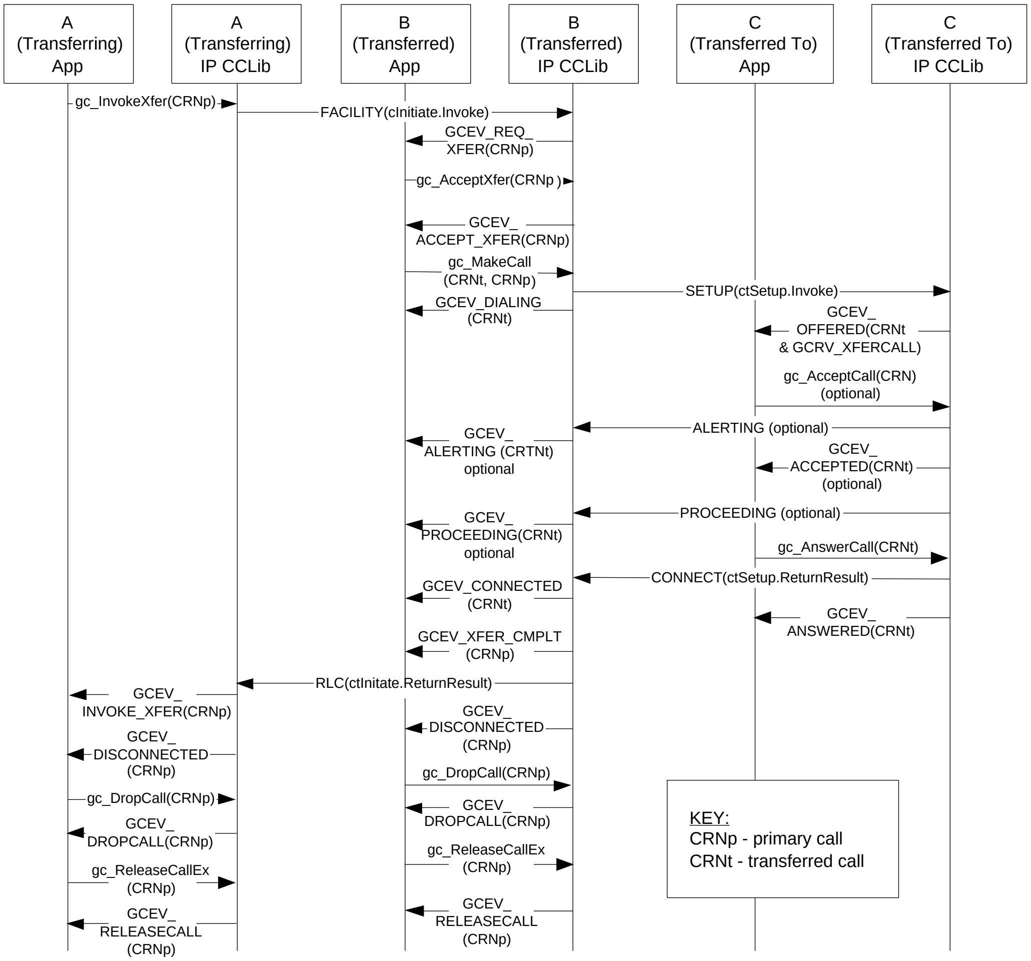
3.2.3 Successful H.450.2 Blind Call Transfer Scenario

As indicated in Figure 10, the precondition for blind transfer is that the transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active (primary) call and are in GCST_CONNNECTED from the perspective of the Global Call API. Completion of a successful blind transfer results in the eventual termination of the primary call, and the creation of the transferred call. Note that the connection of the transferred call is not a mandate for the completion of a blind transfer. It is always possible that the transferred call itself may possibly be left unanswered after ringing (ALERTING indication) and eventually abandoned and still be considered a *successful* blind transfer from the perspective of the transferring endpoint (party A). Successful blind transfer, in this regard requires only that some response notification was received, that is, either ALERTING or CONNECT, from the transferred-to endpoint.

For simplification purposes, Figure 10 does not indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

Figure 10. Successful H.450.2 Blind Call Transfer

Pre condition: Primary call between A and B is connected (not shown).



Post condition: Transferred call between B and C offered.
Primary call between A and B dropped and released.

3.2.4 Unsuccessful H.450.2 Blind Call Transfer Scenarios

There are a several of scenarios where a blind call transfer may fail. The most common scenarios are described in the following topics:

- Party B Rejects Transfer
- No Response From Party B
- No Response From Party C
- Party B Clears Primary Call Before Transfer is Completed
- Party C is Busy When Transfer Attempted

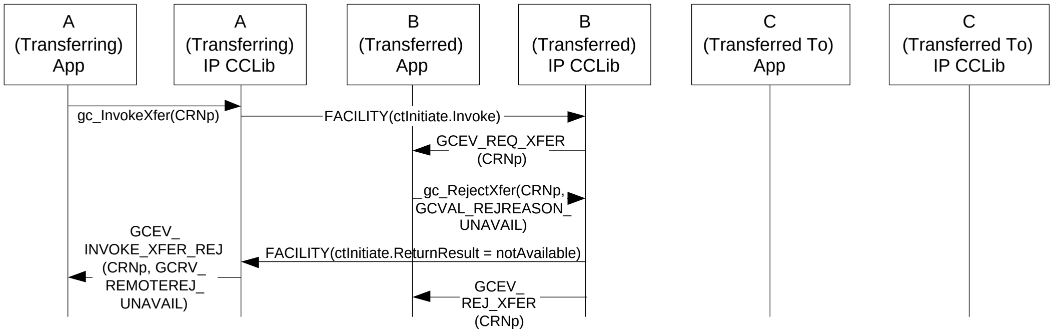
For simplification purposes, none of the following figures indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

3.2.4.1 Party B Rejects Transfer

As indicated in Figure 11, the application at the transferred endpoint (party B) may call the **gc_RejectXfer( )** function to signal via the ctInitiate.ReturnResult APDU that it cannot participate in a transfer. As a result, the GCEV_INVOKE_XFER_REJ termination event is received at transferring endpoint (party A) and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 11. H.450.2 Blind Call Transfer Failure - Party B Rejects Call Transfer

Pre condition: Primary call between A and B is connected (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer(CRNp)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
gc_RejectXfer(CRNp, GCVAL_REJREASON_ UNAVAIL)
GCEV_ INVOKE_XFER_REJ (CRNp, GCRV_ REMOTEREJ_ UNAVAIL)
FACILITY(ctInitiate.ReturnResult = notAvailable)
GCEV_ REJ_XFER (CRNp)

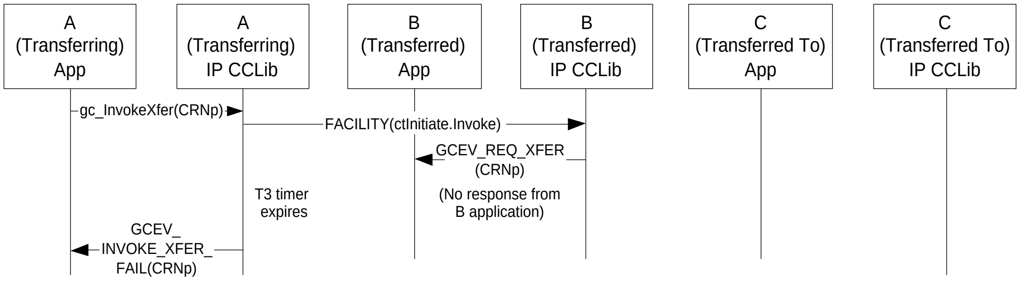
Post condition: Parties A and B remain connected.

3.2.4.2 No Response From Party B

As indicated in Figure 12, the transferred endpoint (party B) may not respond to the ctInitiate.ReturnResult APDU which would cause the T3 timer configured as 20 seconds at the transferring endpoint (party A) to expire. As a result, the GCEV_INVOKE_XFER_FAIL termination event would be received at transferring endpoint (party A) and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 12. H.450.2 Blind Call Transfer Failure - No Response from Party B

Pre condition: Primary call between A and B is connected (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer(CRNp)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
T3 timer expires
(No response from B application)
GCEV_ INVOKE_XFER_ FAIL(CRNp)

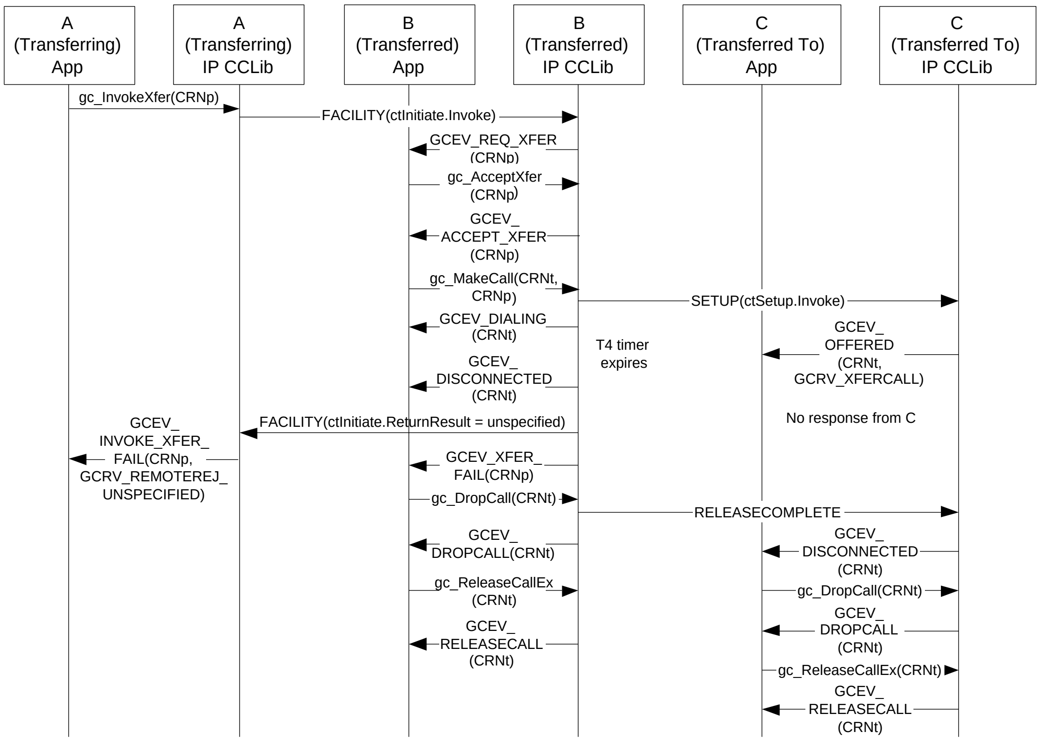
Post condition: Parties A and B remain connected.

3.2.4.3 No Response From Party C

As indicated in Figure 13, the transferred-to endpoint (party C) may not respond to the incoming call which would cause the T4 timer configured as 20 seconds at the transferred endpoint (party B) to expire. As a result, the transferred endpoint (party B) receives the GCEV_DISCONNECT event for the transferred call timeout and after sending a ctInitiate.ReturnResult = Unspecified APDU receives the GCEV_XFER_FAIL event on the primary call. Upon receiving the ctInitiate.ReturnResult = Unspecified APDU, the transferring endpoint (party A) is notified by the GCEV_INVOKE_XFER_FAIL termination event and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 13. H.450.2 Blind Call Transfer Failure - No Response from Party C

Pre condition: Primary call between A and B is in connected (not shown).



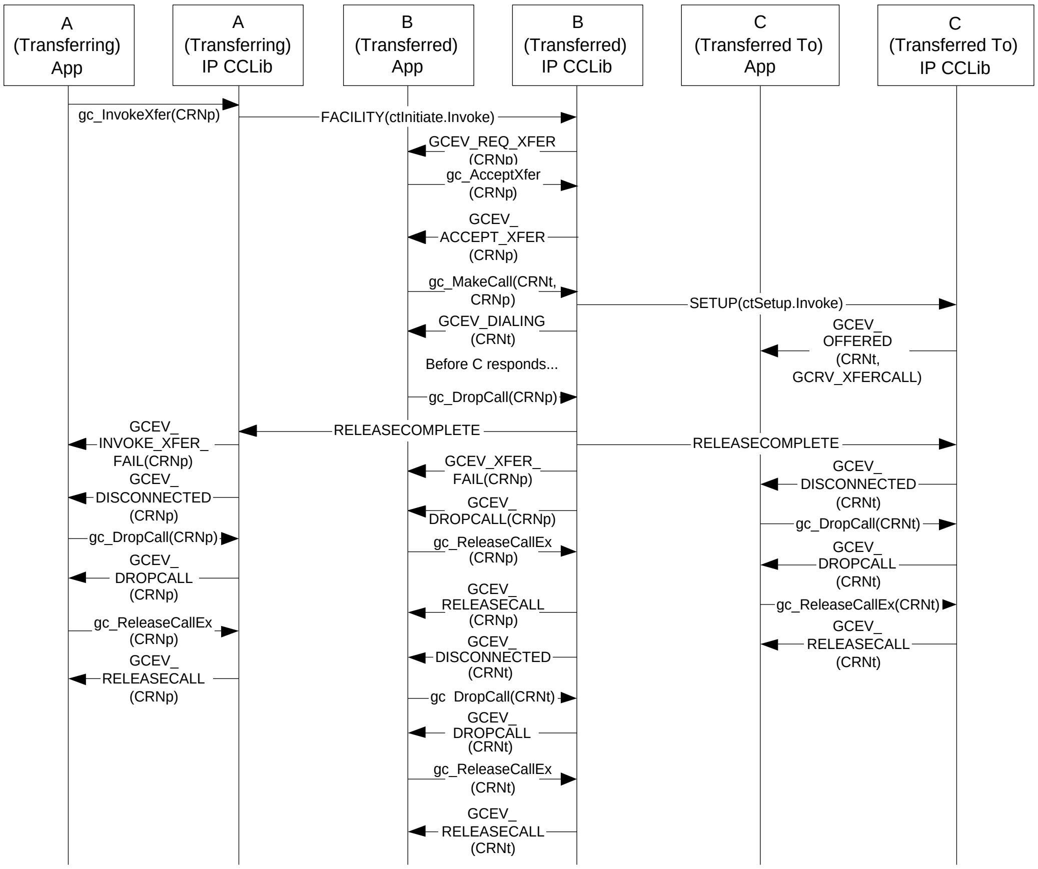
Post condition: Parties A and B remain connected.

3.2.4.4 Party B Clears Primary Call Before Transfer is Completed

The primary call may be cleared at any time while a blind transfer is in progress. As indicated in Figure 14, the transferred endpoint (party B) may clear the primary call while awaiting acknowledgement from the transferred-to endpoint (party C). As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at transferring endpoint (party A) followed by a GCEV_DISCONNECT. Similarly, the GCEV_XFER_FAIL termination event is received at the transferred endpoint (party B) followed by a GCEV_DROPCALL. At this point party A must drop and release the call while party B must release the call. The transferred call will also be abandoned implicitly per H.450.2 once the primary call is abandoned. The transferred-to endpoint will receive the GCEV_DISCONNECTED event at which point it must explicitly drop and release the abandoned transferred call attempt. Note that if instead party A initiated the clearing of the primary call while blind transfer is in progress, the only major difference with the corollary is that party B and not A would react to the primary disconnect (in the same manner as before) and explicitly drop the primary call; otherwise, the behavior is identical.

Figure 14. H.450.2 Blind Call Transfer Failure - Party B Clears Primary Call Before Transfer is Completed

Pre condition: Primary call between A and B is in connected (not shown).



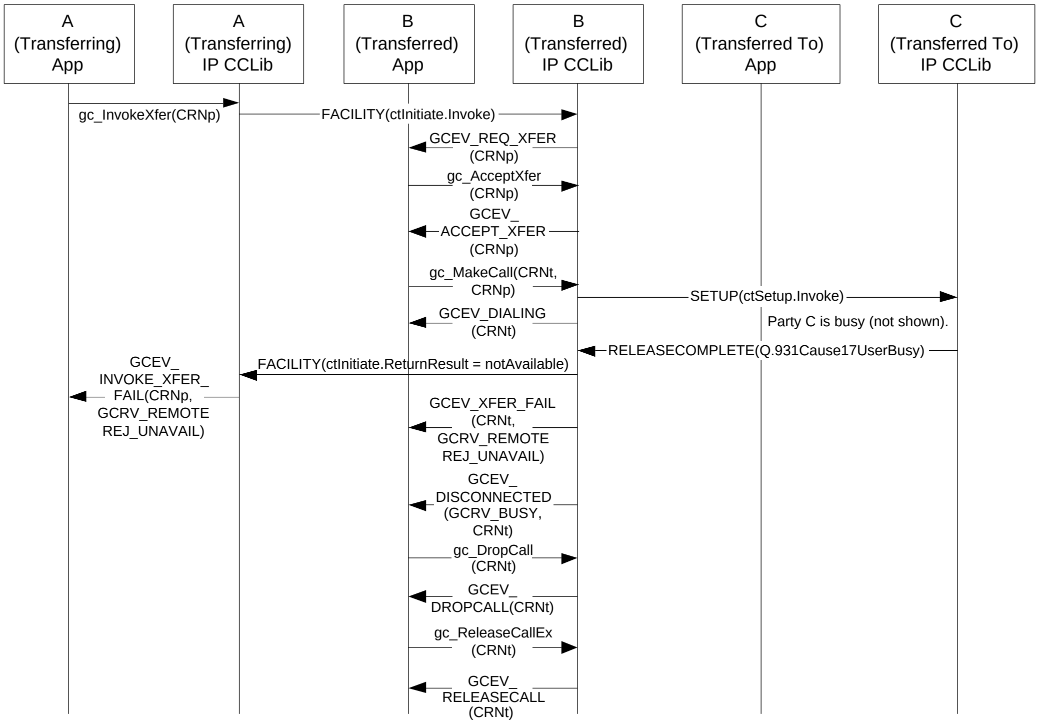
Post condition: Both primary and transfered calls are dropped and released.

3.2.4.5 Party C is Busy When Transfer Attempted

The transferred-to endpoint (party C) may also be busy at the time of transfer (unknown to the transferring endpoint). As indicated in Figure 15, this would result in a Release Complete message with Q.931 Cause 17, User Busy, being returned to the transferred endpoint (party B) indicated to its application via a GCEV_DISCONNECTED event with a cause of GCRV_BUSY. The transferred endpoint (party B) returns a ctInitiate.ReturnError APDU to the transferring endpoint to indicate the transfer failure and in turn must drop the transferred call attempt. As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at transferring endpoint (party A) and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 15. H.450.2 Blind Call Transfer Failure - Party C is Busy When Transfer Attempted

Pre condition: Primary call between A and B is in connected (not shown).
Party C has call connected to another party (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer(CRNp)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
gc_AcceptXfer (CRNp)
GCEV_ ACCEPT_XFER (CRNp)
gc_MakeCall(CRNt, CRNp)
SETUP(ctSetup.Invoke)
GCEV_DIALING (CRNt)
Party C is busy (not shown).
RELEASECOMPLETE(Q.931Cause17UserBusy)
FACILITY(ctInitiate.ReturnResult = notAvailable)
GCEV_ INVOKE_XFER_ FAIL(CRNp, GCRV_REMOTE REJ_UNAVAIL)
GCEV_XFER_FAIL (CRNt, GCRV_REMOTE REJ_UNAVAIL)
GCEV_ DISCONNECTED (GCRV_BUSY, CRNt)
gc_DropCall (CRNt)
GCEV_ DROPCALL(CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_ RELEASECALL (CRNt)

Post condition: Parties A and B remain connected.
Party C also remains connected (to another party not shown).

3.2.5 Endpoint Behavior in H.450.2 Supervised Call Transfer

This section describes the behavior of each of the three endpoints in a supervised call transfer under H.450.2. The assumed preconditions for supervised call transfer are:

- The transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active call, known as the primary call. From the perspective of the Global Call API, party A and party B are both in the GCST_CONNNECTED state.
- The transferring endpoint and the transferred-to endpoint (party C) are participating in an active call, known as the secondary or consultation call. From the perspective of the Global Call call control library, party A and party C are both in the GCST_CONNNECTED state. If party C uses Global Call and is not in the connected state when the transfer is invoked, it may fail to receive the Global Call event for the transfer request (GCEV_REQ_INIT_XFER), which will cause it to receive a GCEV_TASKFAIL.

3.2.5.1 Transferring Endpoint (Party A) Role

As in the case of blind transfer, the transferring endpoint initiates the supervised transfer by calling the **gc_InvokeXfer( )** function. The distinction between blind and supervised transfer usage is the addition of the CRN of the secondary (consultation) call. Calling the **gc_InvokeXfer( )** function at the transferring endpoint with two CRN values results in the sending of an ctIdentify.Invoke APDU in a FACILITY message to the transferred-to endpoint (party C). Once a positive acknowledgement from the transferred-to endpoint is received via a ctIdentify.ReturnResult APDU in a FACILITY message, the transferring endpoint automatically proceeds to invoke the actual call transfer by sending an ctInitiate.Invoke APDU in a FACILITY message to the transferred endpoint (party B).

From this point forward, from the perspective of this endpoint, the behavior is similar to that of a blind or unsupervised transfer. The one difference is that the secondary, consultation call is disconnected once the transferred call is answered at the transferred-to endpoint (party C) and must be explicitly dropped and released. Note however, if the transferred call goes unanswered or fails, the secondary call is left active and maintained in the GCST_CONNECTED state.

3.2.5.2 Transferred Endpoint (Party B) Role

The endpoint to be transferred (party B) has no knowledge of the origins of the destination address information a priori in that it was retrieved as a result of a consultation call. Thus, from the perspective of this endpoint, the behavior and handling of supervised transfer is exactly the same as that of blind transfer.

3.2.5.3 Transferred-To Endpoint (Party C) Role

At any point in time after a secondary consultation call is answered by the transferred-to endpoint, a FACILITY(ctIdentify.Invoke) request may arrive requesting whether it is able to participate in an upcoming transfer as signified by the GCEV_REQ_INIT_XFER event. Assuming that the endpoint is capable, the application calls the **gc_AcceptInitXfer( )** function to accept the transfer along with the intended rerouting number address in the **reroutinginfop** GC_REROUTING_INFO pointer parameter. The IP CCLIB internally returns a newly created callIdentity for the transferred call to use.

From this point forward, the behavior in handling the incoming transferred call from the perspective of this endpoint is identical to that of a blind or unsupervised transfer. The only difference is that the secondary, consultation call is disconnected once the transferred call is answered and must be explicitly dropped and released.

3.2.6 Successful H.450.2 Supervised Call Transfer Scenario

As indicated in Figure 16, the first precondition for supervised H.450.2 transfer is that the transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active call (the primary call) and from the Global Call perspective, in the GCST_CONNECTED state.

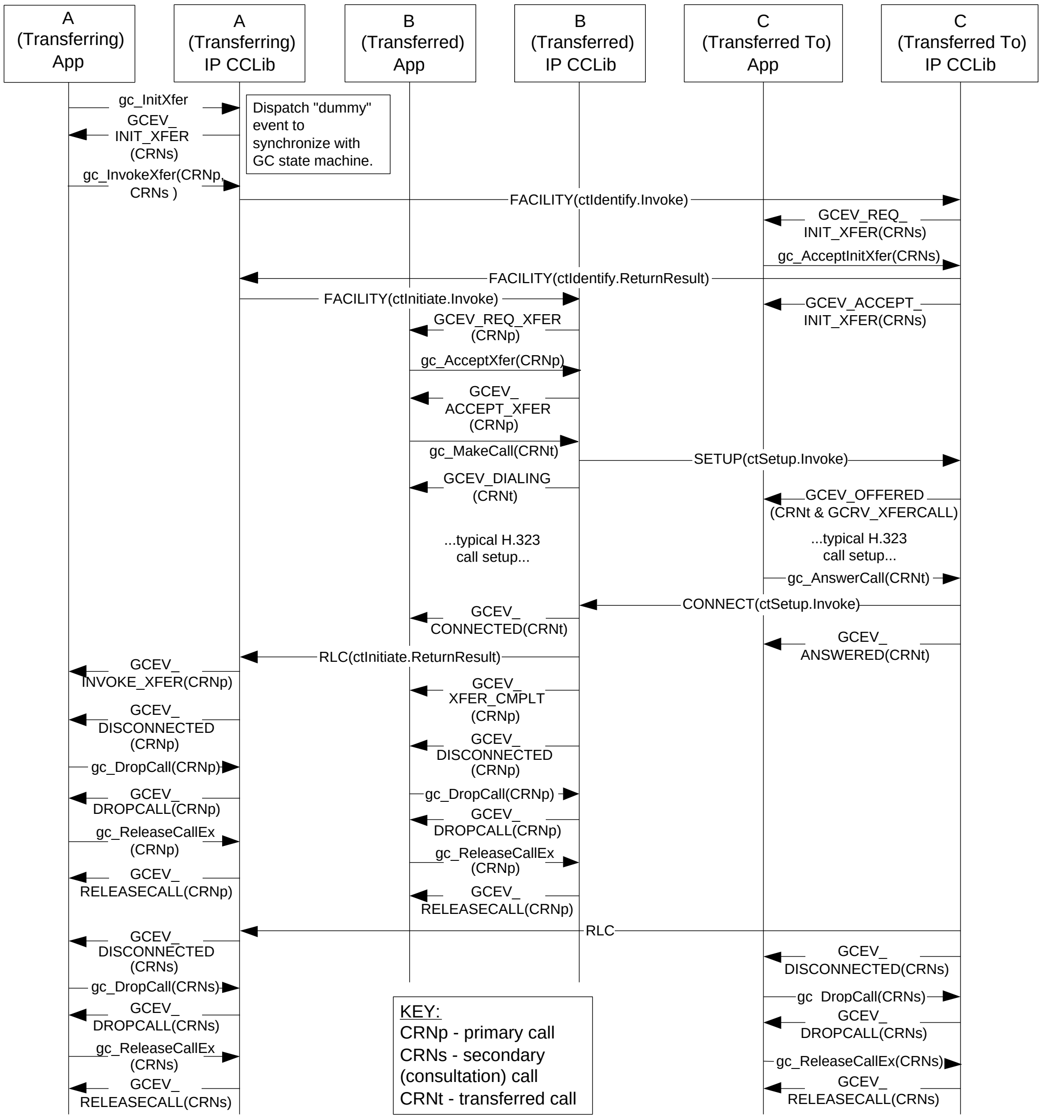
The second precondition for supervised transfer is that a secondary call (the consultation call) from the transferring endpoint (party A) to the transferred-to endpoint (party C) is active and both endpoints are in the GCST_CONNECTED state.

Completion of a successful supervised transfer results in the eventual termination of the primary and secondary (consultation) calls, and the creation of the transferred call. Note that the connection of the transferred call is not a mandate for supervised call transfer. While less likely due to the typical human dialogue on a secondary (consultation) call, it is always possible that the transferred call itself may be left unanswered and eventually abandoned and still be considered a *successful* transfer from the signaling perspective of the transferring endpoint (party A).

For simplification purposes Figure 16 does not indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

Figure 16. Successful H.450.2 Supervised Call Transfer

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).



Post condition: Transferred call between B and C offered (optional whether connected or not).
Primary call between A and B dropped and released.
Secondary (consultation) call between A and C dropped and released.

3.2.7 Unsuccessful H.450.2 Supervised Transfer Scenarios

Note: The same failures that can potentially occur in blind transfer, may take place in supervised transfer as well. See Section 3.2.4, "Unsuccessful H.450.2 Blind Call Transfer Scenarios", on page 54 for more information. The difference being that the secondary, consultation may optionally be cleared as specified in H.450.2.

There are a several other scenarios where a supervised call transfer may fail. The most common scenarios are described in the following topics:

- Party C Timeout
- Party C Rejects the Transfer Request
- Party B Rejects the Transfer Request
- Party B Timeout

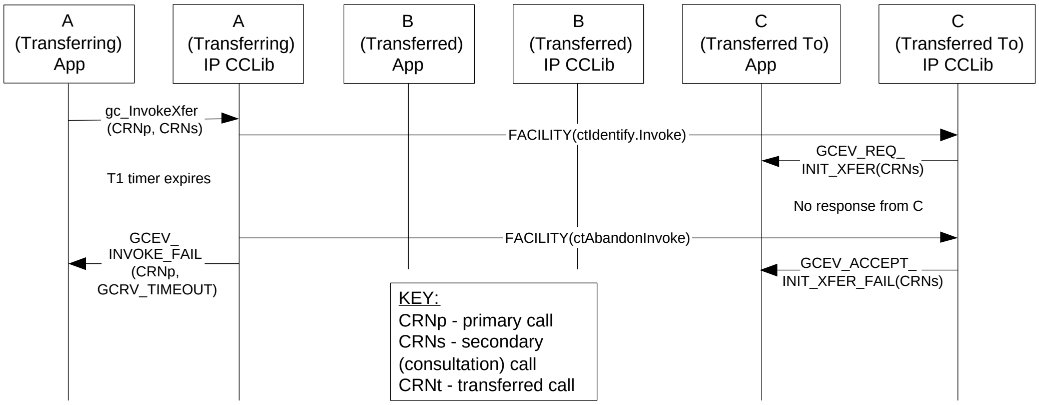
For simplification purposes, none of the following figures indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

3.2.7.1 Party C Timeout

As indicated in Figure 17, the user or application at the transferred-to endpoint (party C) may fail to respond to the ctIdentifiy.Invoke request causing the timer 1 to expire at the transferring endpoint (party A) resulting in an abandoned transfer attempt. As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary).

Figure 17. H.450.2 Supervised Call Transfer Failure - Party C Timeout

Pre condition: Primary call between A and B is connected.
Secondary (consultation) callbetween A and C is connected (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
GCEV_REQ_ INIT_XFER(CRNs)
T1 timer expires
No response from C
FACILITY(ctAbandonInvoke)
GCEV_ INVOKE_FAIL (CRNp, GCRV_TIMEOUT)
GCEV_ACCEPT_ INIT_XFER_FAIL(CRNs)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call

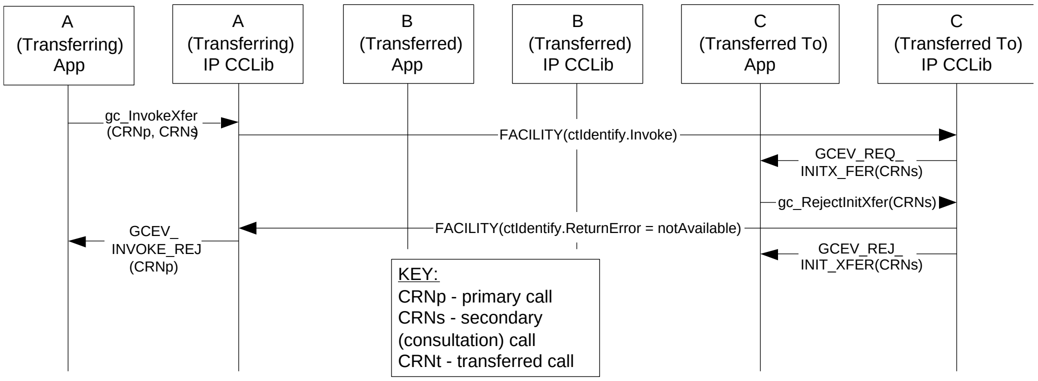
Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.
Transferred call between B and C never initiated.

3.2.7.2 Party C Rejects the Transfer Request

As indicated in Figure 17, the user or application at the transferred-to endpoint (party C) may call the **gc_RejectInitXfer( )** function to signal via the ctInIdentify.ReturnResult APDU that it cannot participate in a transfer. As a result, the GCEV_INVOKE_XFER_REJ termination event is received at the transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary); GCST_CONNECTED state from the perspective of both A and B.

Figure 18. H.450.2 Supervised Call Transfer Failure - Party C Rejects the Transfer Request

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).



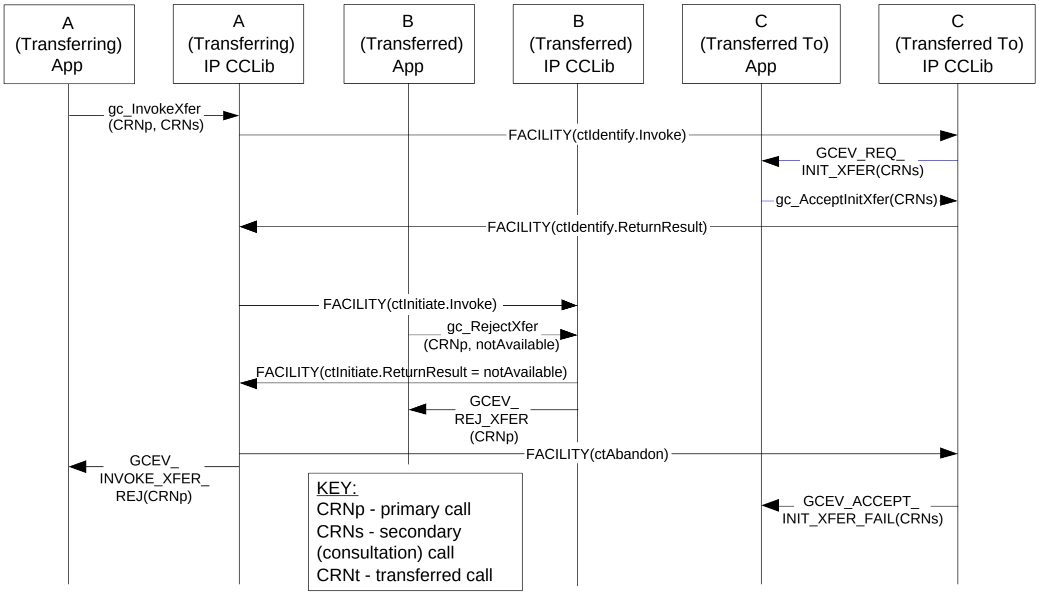
Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.

3.2.7.3 Party B Rejects the Transfer Request

As indicated in Figure 19, the user or application at the transferred endpoint (party B) may call the **gc_RejectXfer( )** function to reject the transfer request and signal via the ctInitiate.ReturnResult APDU that it cannot participate in a transfer. As a result, the GCEV_INVOKE_XFER_REJ termination event is received at transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary); GCST_CONNECTED state from the perspective of both A and B.

Figure 19. H.450.2 Supervised Call Transfer Failure - Party B Rejects the Transfer Request

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
GCEV_REQ_ INIT_XFER(CRNs)
gc_AcceptInitXfer(CRNs)
FACILITY(ctIdentify.ReturnResult)
FACILITY(ctInitiate.Invoke)
gc_RejectXfer (CRNp, notAvailable)
FACILITY(ctInitiate.ReturnResult = notAvailable)
GCEV_ REJ_XFER (CRNp)
FACILITY(ctAbandon)
GCEV_ INVOKE_XFER_ REJ(CRNp)
GCEV_ACCEPT_ INIT_XFER_FAIL(CRNs)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call

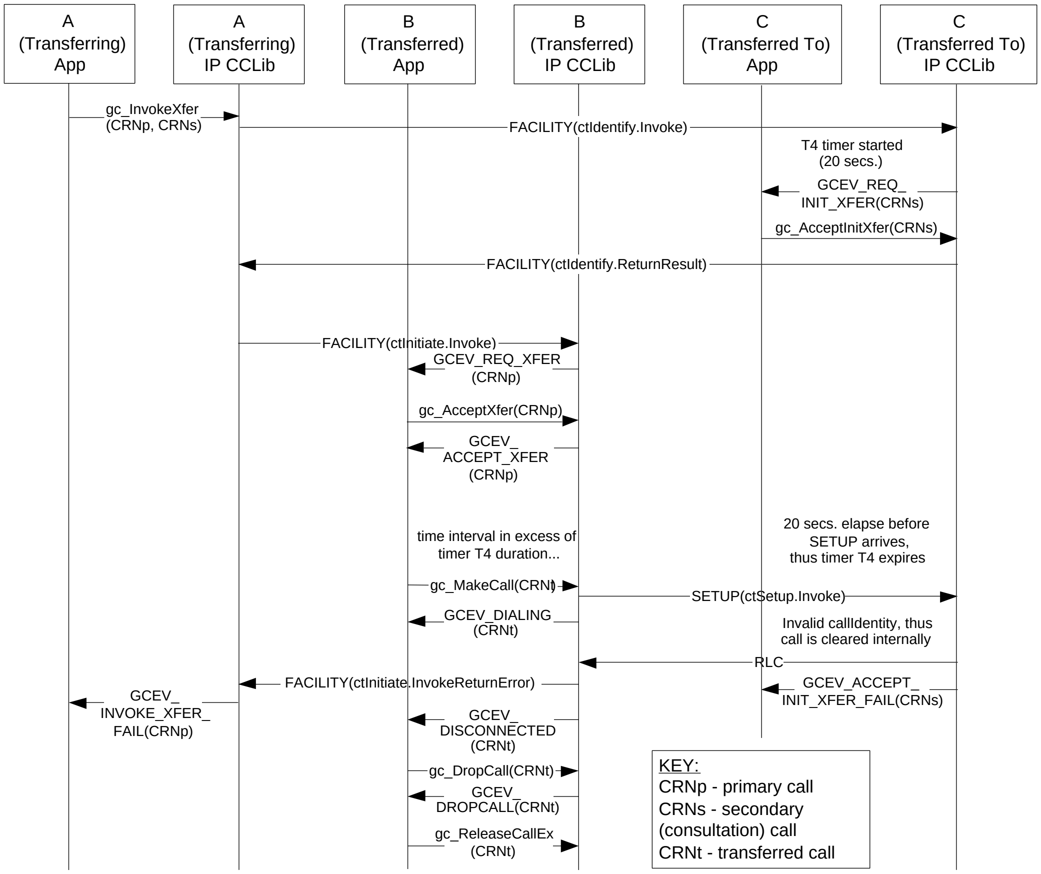
Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.

3.2.7.4 Party B Timeout

As indicated in Figure 20, the user or application at the transferred-to endpoint (party C) may receive the transferred call after the T4 timer expires. If this is the case and the callIdentity is cleared as a result of the T4 expiry, the transferred-to endpoint will clear or reject the transferred call as indicated by a GCEV_DISCONNECTED event at the transferred endpoint (party B) and a GCEV_INVOKEXFER_FAIL event at the transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary); GCST_CONNECTED state from the perspective of both A and B.

Figure 20. H.450.2 Supervised Call Transfer Failure - Party B Timeout

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
T4 timer started (20 secs.)
GCEV_REQ_ INIT_XFER(CRNs)
gc_AcceptInitXfer(CRNs)
FACILITY(ctIdentify.ReturnResult)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
gc_AcceptXfer(CRNp)
GCEV_ ACCEPT_XFER (CRNp)
time interval in excess of timer T4 duration...
20 secs. elapse before SETUP arrives, thus timer T4 expires
gc_MakeCall(CRNt)
SETUP(ctSetup.Invoke)
GCEV_DIALING (CRNt)
Invalid callIdentity, thus call is cleared internally
RLC
FACILITY(ctInitiate.InvokeReturnError)
GCEV_ACCEPT_ INIT_XFER_FAIL(CRNs)
GCEV_ INVOKE_XFER_ FAIL(CRNp)
GCEV_ DISCONNECTED (CRNt)
gc_DropCall(CRNt)
GCEV_ DROPCALL(CRNt)
gc_ReleaseCallEx (CRNt)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call

Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.

3.3 Call Transfer Scenarios When Using SIP

The Global Call API functions that supports IP call transfer are described in the *Global Call API Library Reference*; protocol-specific information about the individual call transfer APIs can be found in the subsections of Section 7.3, "Global Call Function Variances for IP". General information on implementing call transfer can be found in Section 4.25, "Call Transfer", on page 237, and SIP-specific details can be found in Section 4.25.5, "Call Transfer When Using SIP", on page 242.

The following topics describe the call transfer capabilities provided when using the SIP call transfer supplementary service:

- General Conditions for SIP Call Transfers
- Endpoint Behavior in Unattended SIP Call Transfers
- Successful Unattended SIP Call Transfer Scenarios
- Endpoint Behavior in Attended SIP Transfers
- Successful SIP Attended Call Transfer Scenarios
- Unsuccessful Call Transfer Scenarios

3.3.1 General Conditions for SIP Call Transfers

SIP call transfer uses the REFER method (with NOTIFY support) to reroute a call (a SIP dialog) after the call has been established; in other words, after two endpoints have an established media path.

There are two fundamental types of call transfer:

- Unattended transfer, which is referred to as "blind transfer" in most other technologies and protocols. In this type of transfer the transferring party (called the Transferor in SIP) has a call (or SIP dialog) with the transferred party (called the Transferee in SIP) but not with the transferred-to party (called the Transfer Target in SIP).
- Attended transfer, which is referred to as "supervised transfer" in most other technologies and protocols. In this type of transfer, the Transferor has a dialog with both the Transferee and the Transfer Target.

In its simplest terms, a SIP call transfer involves the Transferor issuing a REFER to the Transferee to cause the Transferee to issue an INVITE to the Transfer Target. The Transferee and Transfer Target negotiate the media without regard to the media that had been negotiated between the Transferor and the Transferee, just as if the Transferee had initiated the INVITE on its own.

Once a transfer request is accepted by the Transferee, the Transferor is not allowed to send another transfer request to the Transferee. If a transfer request is rejected or fails, Transferor is allowed to attempt another transfer request to Transferee again.

The disposition of the media streams between the Transferor and the Transferee is not altered by the REFER method. A successful REFER transaction does not terminate the session between the Transferor and the Transferee; if those parties wish to terminate their session, they must do so with a subsequent BYE request.

In the SIP call transfer protocol the Transferor is notified when the Transferee accepts the REFER transfer request. The Global Call Library allows this notification to be signaled to the application as a GCEV_INVOKE_XFER_ACCEPTED event. This event is optional, and is disabled (or masked) by default. The party A application can enable and disable this event at any time after the line device is opened using the **gc_SetConfigData( )** function. See Section 4.25.5.1, "Enabling GCEV_INVOKE_XFER_ACCEPTED Events", on page 242 for details.

When performing a call transfer operation, all involved call handles must be on the same stack instance. This imposes the following application restrictions for call transfer operations

- When performing an attended call transfer at party A, both the consultation line device and the transferring line device must be on the same virtual board.
- When performing a call transfer (either attended or unattended) at party B, both the transferring line device and the transferred line device must be on the same virtual board.
- When performing an attended call transfer at party C, both the consultation line device and the transferred-to line device must be on the same virtual board.

Interoperability Issues

The latest standards for the SIP REFER method are defined in IETF RFC 3515, published in April 2003. Many existing implementations of REFER, including the current Global Call implementation, are based on the previous draft of the REFER method and are not fully compliant with RFC 3515. The most significant non-compliance issues are:

- no initial NOTIFY after sending out 202 accept to REFER request
- no subscription state information in NOTIFY message
- no NOTIFY generated by the Transferee (Transferred party) after the call is terminated
- any NOTIFY received by the Transferor (Transferring party) after the call is terminated is treated as a stand-alone message outside of any call; such messages may be sent by the Transferee as a late retry if a 200 OK to NOTIFY was lost before the call was terminated, or by a non-Global Call User Agent

3.3.2 Endpoint Behavior in Unattended SIP Call Transfers

The precondition for unattended call transfer (commonly referred to as "blind call transfer" in other technologies and protocols) is that the transferring endpoint (party A, or Transferor in SIP terminology) and the transferred endpoint (party B or Transferee in SIP terms) are participating in an active call, known as the primary call. From the perspective of the Global Call API, both parties are in the GCST_CONNNECTED state. Completion of a successful unattended transfer results in the eventual termination of the primary call, and the creation of the transferred call between party B and the Transfer Target (party C).

3.3.2.1 Transferor or Transferring Endpoint (party A)

The Transferor (party A) initiates an unattended transfer by calling the **gc_InvokeXfer( )** function on the CRN of the primary call (CRNp), which results in the sending a REFER message to the Transferee (party B). The Refer-To header in the REFER request is constructed from either the char *numberstr or the GC_MAKECALL_BLK *makecallp parameter in the **gc_InvokeXfer( )**

function, following the same rules as **gc_MakeCall( )**. The Referred-By header is automatically constructed with the local URI—the same as the From or To header, depending on the direction of the initial call INVITE. Optionally, the Transferor can override the default Referred-By header by inserting a Referred-By header in the **gc_InvokeXfer( )** parm block. Party A will be notified if REFER is accepted or rejected by transferred endpoint (party B).

If party A receives a 2xx response to the REFER (indicating that is was accepted by party B), a GCEV_INVOKE_XFER_ACCEPTED event may optionally be generated. This optional event is disabled by default; the event can be enabled or disabled at any time after the line device has been opened by use of the **gc_SetConfigData( )** function.

The primary call may be terminated by either party before transferred call is completed. Note that in an H.450.2 implementation, party A will actually get INVOKE_XFER_REJ event locally if party A terminates the primary call before receiving final status from party B. Unlike an H.450.2 transfer, party A in a SIP transfer will **not** get any transfer termination event if party A terminates the primary call before receiving final status from party B. This is because there is no way to be sure if the transfer is successful or if it failed and it is party A's responsibility to update the application transfer states in this case. This is a common scenario in blind transfer where party A does not care about the transferred call status and drops the primary call immediately after receiving a INVOKE_XFER_ACCEPTED event.

After the primary call is terminated, Global Call treats any received NOTIFY message as a stand-alone message outside any call. In this case, the library generates a GCEV_EXTENSION event to notify the application of the message, and the application has to respond to the NOTIFY (accepting it or rejecting it) using the **gc_Extension( )** function as described in Section 4.12.8, "Responding to NOTIFY Requests", on page 191.

If the primary call remains connected and the REFER subscription is alive, party A **may** be notified of the final status of transferred call from party B. The notification of transferred call status is optional depending on party B.

From party A's perspective, a call transfer is considered successful as long as GCEV_INVOKE_XFER_ACCEPTED (if enabled) and GCEV_INVOKE_XFER events are received. If the optional GCEV_INVOKE_XFER_ACCEPTED event type is enabled, that event is generated by receiving a 2xx response (to the REFER request) from party B. The GCEV_INVOKE_XFER event is generated by receiving from party B a NOTIFY of either 180 Ringing or 2xx final status on the transferred call.

The primary call will be disconnected locally immediately after generating GCEV_INVOKE_XFER event. From the Global Call API perspective, the primary call is terminated at the transferring endpoint as indicated by the GCEV_DISCONNECTED event implying the Transferor endpoint is then responsible for dropping and releasing the primary call.

3.3.2.2 Transferee or Transferred Endpoint (Party B)

The endpoint to be transferred (party B, or Transferee in SIP terms) is notified of the request to transfer from the initiating endpoint via a GCEV_REQ_XFER event on CRNp. If party B accepts the transfer request via **gc_AcceptXfer( )** function call on CRNp, a 202 Accepted response is sent to party A.

Party B retrieves the destination address information from the unsolicited transfer request via the GC_REROUTING_INFO structure passed with the GCEV_REQ_XFER event. Party B uses the rerouting address information (Refer-To address) to initiate a call to the new destination party via **gc_MakeCall( )** on CRNt. From the perspective of the application, this transferred call is treated in the same manner as a normal singular call and the party receives intermediate call state events as to the progress of the call (e.g., GCEV_DIALING, GCEV_ALERTING, GCEV_PROCEEDING, and GCEV_CONNECTED).

If the CRNp number is included during the **gc_MakeCall( )** on CRNt and the primary call is in the connected state, then a GCEV_XFER_CMPLT event is generated on CRNp once the transferred call is connected. If the CRNp number is not included, there will be no notification to the primary call and/or party A of the transferred call status. The CRNp number must not be included in the **gc_MakeCall()** if primary call was disconnected prior to making transferred call.

When party B receives the indication from party C that the call transfer was successful (200 OK), the party B application is notified of the success via a GCEV_XFER_CMPLT event on CRNp. If the primary call is still connected, party B will notify party A of the transfer status (200 OK). Then party B implicitly, without user/application initiation, disconnects the primary call with the party A. Although the primary call to party A is implicitly dropped, the call itself must still be explicitly dropped via **gc_DropCall( )** and released via **gc_ReleaseCallEx( )** to resynchronize the local state machine.

Either the party A or party B application may terminate the primary call after party B accepts the transfer request. If the primary call is terminated by party A before receiving any call transfer termination event (GCEV_INVOKE_XFER or GCEV_INVOKE_XFER_FAIL), party B will not notify party A of the transfer status. If the primary call is terminated by party B before receiving any transferred call provisional or final response from party C, party B *will* send NOTIFY to party A with 200 OK and terminate the REFER subscription before sending BYE to party A.

If the primary call is disconnected before making the transferred call to party C, party B must not include the primary call CRN (CRNp) when making the transferred call to party C. Otherwise, a Global Call error will be returned.

Note that the primary call can be disconnected prior to making the transferred call only during an unattended transfer because the transferred call can be established independently from the primary call. During an attended transfer, the transferred call cannot be established after the primary call is disconnected because the primary call database contains the Replaces information that is required by the transferred call.

If the Referred-By header exists in the REFER message, it is passed to the application via the GCEV_REQ_XFER event if SIP message information access was enabled (by setting the IP_SIP_MSGINFO_ENABLE in the sip_msginfo_mask field of the IP_VIRTBOARD data structure) when the virtual board was started.

3.3.2.3 Transfer Target or Transferred-To Endpoint (Party C)

From the perspective of party C, the transferred call is, for the most part, treated as a typical incoming call. The call is first notified to the application by a GCEV_DETECTED or GCEV_OFFERED event on CRNt. The GCRV_XFERCALL cause value is provided in the event to alert the application that this call offering is the result of a transfer, but only if the incoming

INVITE contains Referred-By or Replaces information indicating a new transferred call. Referred-By and Replaces information, if present, is also attached to GCEV_OFFERED events if SIP header access was enabled (by setting the IP_SIP_MSGINFO_ENABLE value in the sip_msginfo_mask field of the IP_VIRTBOARD data structure) when the virtual board was started.

At that point, the application may retrieve the typical calling party information on CRNt. Party C is then provided the same methods of action as a typical incoming call, namely to alert party B that the call is proceeding (typically for gateways), ringback notification that the local user is being alerted, or simply that the call is answered. The only behavior change from this endpoint over typical non-transferred calls is whether to handle the calling party information any differently because it is the result of a transfer.

3.3.3 Successful Unattended SIP Call Transfer Scenarios

This section describes various scenarios for successful call transfers under the SIP protocol. The scenarios include:

- Successful Transfer with Notification of Connection
- Successful Transfer with Notification of Ringing
- Successful Transfer with Primary Call Cleared Prior to Transfer Completion

All of the scenarios indicate all three common naming conventions for the three parties involved in a call transfer: parties (A, B, and C), endpoints (transferring, transferred, and transferred-to), and SIP roles (Transferor, Transferee, and Transfer Target). “IP CClib” refers to the call control library and SIP stack portions of Global Call. “Non-Global Call” is used to represent a User Agent that might behave legally but differently than Global Call. Pre and post conditions are explicitly listed in each scenario, but the common pre-condition for all scenarios is that the Transferor (party A) and the Transferee (party B) are participating in an active (primary) call and are in the GCST_CONNNECTED state from the perspective of the Global Call API.

For simplification purposes, none of the figures indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related SIP calls.

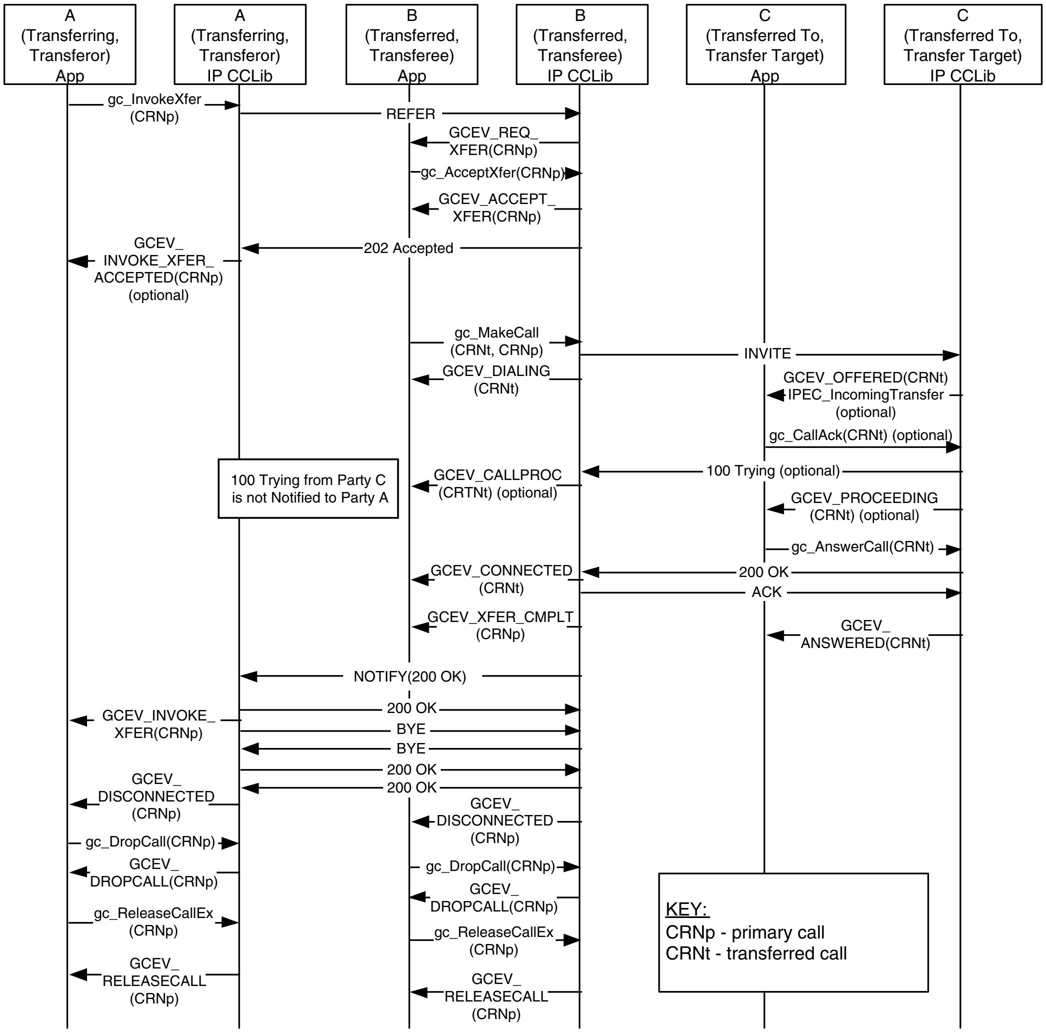
All of the following scenarios illustrate the optional GCEV_INVOKE_XFER_ACCEPTED event, which is disabled by default. The party A application can enable and disable this event at any time after the line device is opened using the **gc_SetConfigData( )** function.

3.3.3.1 Successful Transfer with Notification of Connection

Figure 21 illustrates the basic successful scenario, with party A receiving notification from party B after the transferred call between party B and party C has been connected. The SIP dialog for the primary call between party A and party B is automatically disconnected, and both parties then tear down the call using **gc_DropCall( )** and **gc_ReleaseCallEx( )**.

Figure 21. Successful SIP Unattended Call Transfer, Party A Notified with Connection

Pre condition: Primary call between A and B is connected (not shown).



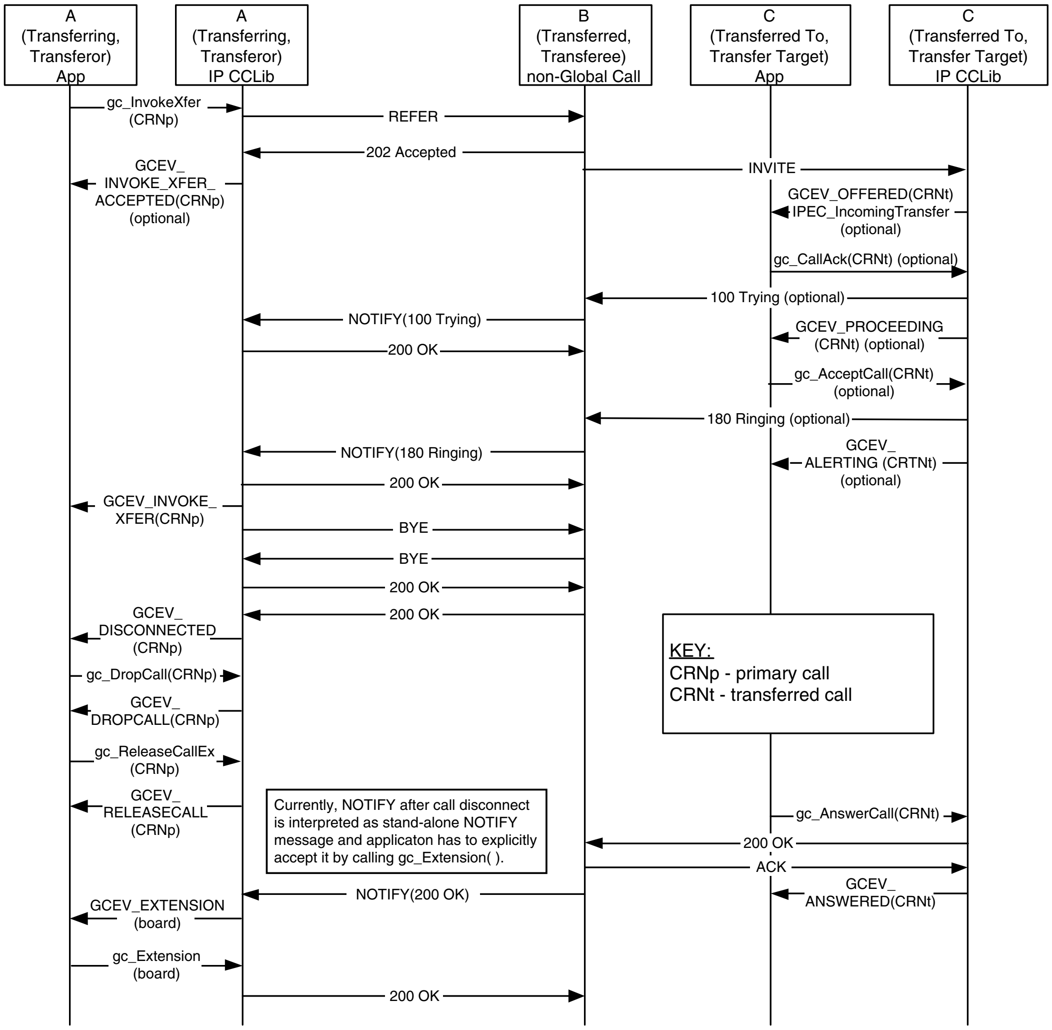
Post condition: Transferred call between B and C offered.
Primary call between A and B dropped and released

3.3.3.2 Successful Transfer with Notification of Ringing

Figure 22 illustrates a valid scenario for which Global Call does not support the party B role. In this scenario, party B notifies party A that the transfer has completed as soon as party C responds to the INVITE with a 100 Trying or 180 Ringing. The Call Control Library at Party A disconnects the primary call with party B after the notification and the application then must tear down the call using **gc_DropCall( )** and **gc_ReleaseCallEx( )**.

Figure 22. Successful SIP Unattended Call Transfer, Party A Notified with Ringing

Pre condition: Primary call between A and B is connected (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) non-Global Call
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
202 Accepted
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp) (optional)
INVITE
GCEV_OFFERED(CRNt) IPEC_IncomingTransfer (optional)
gc_CallAck(CRNt) (optional)
100 Trying (optional)
NOTIFY(100 Trying)
GCEV_PROCEEDING (CRNt) (optional)
200 OK
gc_AcceptCall(CRNt) (optional)
180 Ringing (optional)
NOTIFY(180 Ringing)
GCEV_ ALERTING (CRTNt) (optional)
200 OK
GCEV_INVOKE_ XFER(CRNp)
BYE
BYE
200 OK
200 OK
GCEV_ DISCONNECTED (CRNp)
KEY:
CRNp - primary call
CRNt - transferred call
gc_DropCall(CRNp)
GCEV_ DROPCALL(CRNp)
gc_ReleaseCallEx (CRNp)
GCEV_ RELEASECALL (CRNp)
Currently, NOTIFY after call disconnect is interpreted as stand-alone NOTIFY message and applicaton has to explicitly accept it by calling gc_Extension( ).
gc_AnswerCall(CRNt)
200 OK
ACK
GCEV_ ANSWERED(CRNt)
NOTIFY(200 OK)
GCEV_EXTENSION (board)
gc_Extension (board)
200 OK

Post condition: Transferred call between B and C offered.
Primary call between A and B dropped and released.

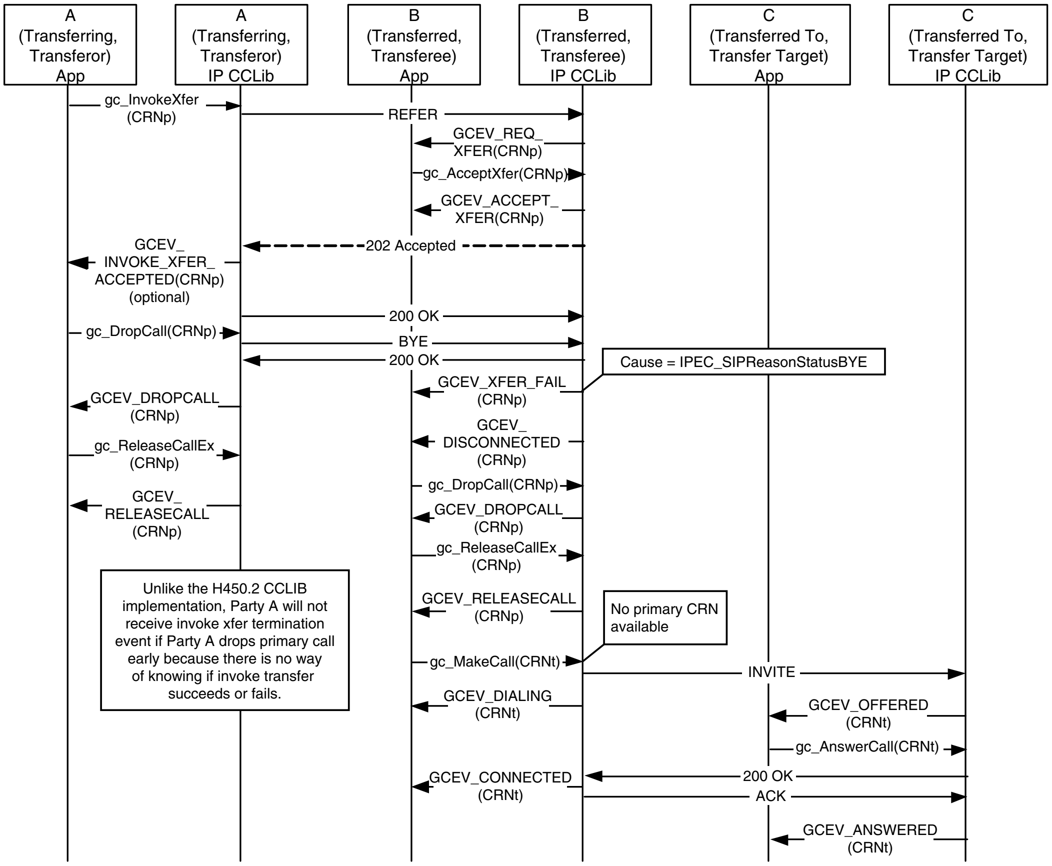
3.3.3.3 Successful Transfer with Primary Call Cleared Prior to Transfer Completion

The SIP protocol supports unattended transfer scenarios where the primary call is cleared or dropped before the transfer completes. In some other technologies and protocols, these scenarios are referred to as “unattended blind transfers” as opposed to “attended blind transfers” where the primary call is maintained until completion. Note that scenarios similar to these are not supported by the H.450.2 protocol.

Figure 23 illustrates a scenario in which party A drops the primary call with party B as soon as it receives notification that party B has accepted the transfer request. In this scenario, party A does not receive any notification that the transfer has completed.

Figure 23. Successful SIP Unattended Call Transfer, Party A Clears Primary Call prior to Transfer Completion

Precondition: Primary call between A and B is connected (not shown).

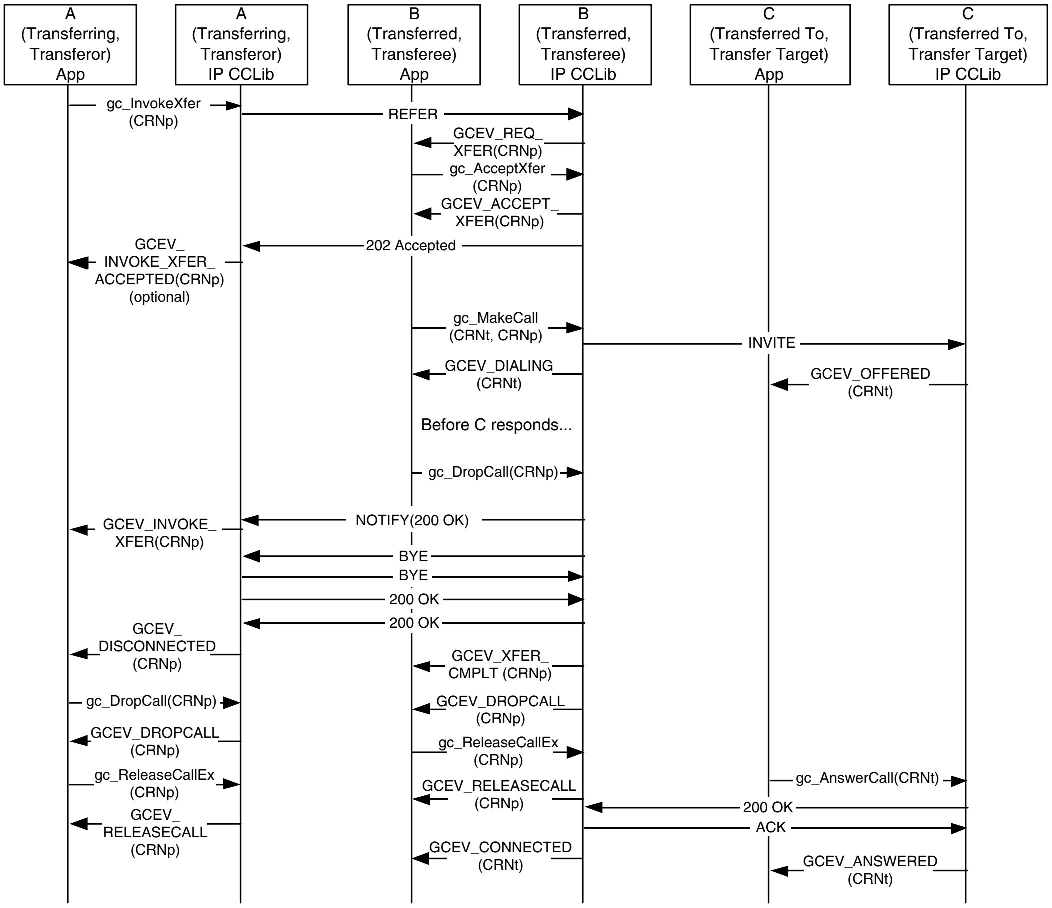

A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_XFER(CRNp)
gc_AcceptXfer(CRNp)
GCEV_ACCEPT_XFER(CRNp)
GCEV_INVOKE_XFER_ACCEPTED(CRNp) (optional)
202 Accepted
200 OK
gc_DropCall(CRNp)
BYE
200 OK
Cause = IPEC_SIPReasonStatusBYE
GCEV_XFER_FAIL (CRNp)
GCEV_DROPCALL (CRNp)
GCEV_DISCONNECTED (CRNp)
gc_ReleaseCallEx (CRNp)
gc_DropCall(CRNp)
GCEV_RELEASECALL (CRNp)
GCEV_DROPCALL (CRNp)
gc_ReleaseCallEx (CRNp)
Unlike the H450.2 CCLIB implementation, Party A will not receive invoke xfer termination event if Party A drops primary call early because there is no way of knowing if invoke transfer succeeds or fails.
GCEV_RELEASECALL (CRNp)
No primary CRN available
gc_MakeCall(CRNt)
INVITE
GCEV_DIALING (CRNt)
GCEV_OFFERED (CRNt)
gc_AnswerCall(CRNt)
GCEV_CONNECTED (CRNt)
200 OK
ACK
GCEV_ANSWERED (CRNt)

Post Condition: Primary call is dropped and released.
Transferred call is connected.

Figure 24 illustrates a scenario in which party B drops the primary call with party A after accepting the transfer request and issuing INVITE to party C, but before receiving any response from party C. In this scenario, party B does notify party A, but this notification only signifies that party B has acted on the transfer request and not that the transfer has actually completed.

Figure 24. Successful SIP Unattended Call Transfer, Party B Clears Primary Call prior to Transfer Completion

Pre condition: Primary call between A and B is connected (not shown).



Post condition: Primary call is dropped and released.
Transferred call is connected.

3.3.4 Endpoint Behavior in Attended SIP Transfers

The assumed preconditions for attended SIP call transfer (commonly referred to as “supervised call transfer” in other technologies and protocols) are:

- The transferring endpoint (party A, or Transferor in SIP terminology) and the transferred endpoint (party B, or Transferee in SIP terms) are participating in an active call, known as the primary call. From the perspective of the Global Call API, party A and party B are both in the GCST_CONNNECTED state.
- The Transferor and the transferred-to party (party C or the Transfer Target in SIP terminology) are participating in an active call, known as the secondary or consultation call. From the perspective of the Global Call call control library, party A and party C are both in the GCST_CONNNECTED state.

Completion of a successful attended transfer results in the eventual termination of the primary and secondary calls, and the creation of the transferred call between party B and the party C.

3.3.4.1 Transferor or Transferring Endpoint (Party A)

SIP does not support or require a transfer initiation process to obtain the rerouting number as in H.323/H.450.2 supervised transfer. To be consistent with the generic Global Call supervised transfer scenario, the party A application in a SIP attended transfer can call **gc_InitXfer( )**, but no request / response messages will be exchanged between party A and party C as a result. Following this function call, party A always receives a GCEV_INIT_XFER completion event with a dummy rerouting address. To alert party C of incoming transfer process, party A can only notify party C by application data or human interaction outside of SIP protocol.

Just as in the case of unattended transfers, an attended transfer is actually initiated when the Transferor calls the **gc_InvokeXfer( )** function. The difference between unattended and attended transfer usage is the inclusion of the CRN of the secondary (consultation) call as a parameter in the function call. When the Transferor calls **gc_InvokeXfer( )** with two CRN values, a REFER message with a replace parameter in the Refer-To header is sent to the Transferee (party B).

From this point onward, the behavior at this endpoint is similar to that of a unattended transfer, except that the application must also drop the secondary/consultation call at transfer completion. Unlike H.450.2, Global Call will not disconnect the secondary/consultation call once the transferred call is answered at party C.

Because SIP does not require any pre-invocation setup for attended call transfers, the Transferor (party A) can actually treat either of the two active calls as the primary call, and can send the REFER to either of the remote endpoints. This fact provides a recovery mechanism in case one of the remote endpoints does not support the REFER method, as illustrated in the scenarios in the following section.

Protecting and Exposing the Transfer Target

The ability to direct the REFER to either of the parties to which the Transferor provides the opportunity to protect the Transfer Target.

To protect the Transfer Target, the Transferor simply reverses the primary and secondary call CRNs when calling **gc_InvokeXfer( )** to reverse the roles of the two remote parties. The original Transfer Target will now send INVITE to the original Transferee, so that the Transferee is effectively "called back" by the Transfer Target. This has the advantage of hiding information about the original Transfer Target from the original transferee, although the Transferee's experience in this scenario will be different that in current systems PBX or Centrex systems.

To expose the Transfer Target and provide an experience similar to current PBX and Centrex systems, the Transferor uses the secondary call to alert the Transfer Target to the impending transfer, but then disconnects the secondary call and completes the transfer as an unattended transfer. In this case, the **gc_InvokeXfer( )** call only includes the CRN of the primary call.

3.3.4.2 Transferee or Transferred Endpoint (Party B)

This endpoint behaves in the same manner as in unattended transfer with one exception: the INVITE that is sent from Party B to Party C for the transferred call contains a Replaces header that is obtained from the replace parameter in the Refer-To header of the REFER from Party A.

Note that the primary call cannot be disconnected prior to making the transferred call during an attended transfer because the primary call database contains the Replaces information that is required to establish the transferred call.

3.3.4.3 Transfer Target or Transferred-To Endpoint (Party C)

This endpoint behaves in much the same manner as in an unattended transfer with one additional feature and one additional responsibility.

If the Replaces header exists in the incoming INVITE, Global Call automatically matches the Replaces value with any existing connected call on Party C. If a matching call (the secondary or consultation call) is found, that call's CRNs is passed to the application as a GCPARM_SECONDARYCALL_CRN parameter in the GC_PARM_BLK that is attached to the GCEV_OFFERED event.

The party C application must also drop the secondary/consultation call when the transfer completes. Unlike H.450.2 call transfer, Global Call does not automatically disconnect the secondary call once the transferred call answered at the party C.

3.3.5 Successful SIP Attended Call Transfer Scenarios

This section describes the basic scenario for successful SIP call transfer and the scenarios for recovery from two conditions that can block transfer completion. The scenarios include:

- Successful SIP Attended Call Transfer
- Attended Transfer when REFER is Not Globally Supported
- Attended Transfer When Contact URI is Not Globally Routable

The scenarios all illustrate the optional GCEV_INVOKE_XFER_ACCEPTED event, which is disabled by default. The Transferor application can enable and disable this event at any time after the line device is opened using the **gc_SetConfigData( )** function.

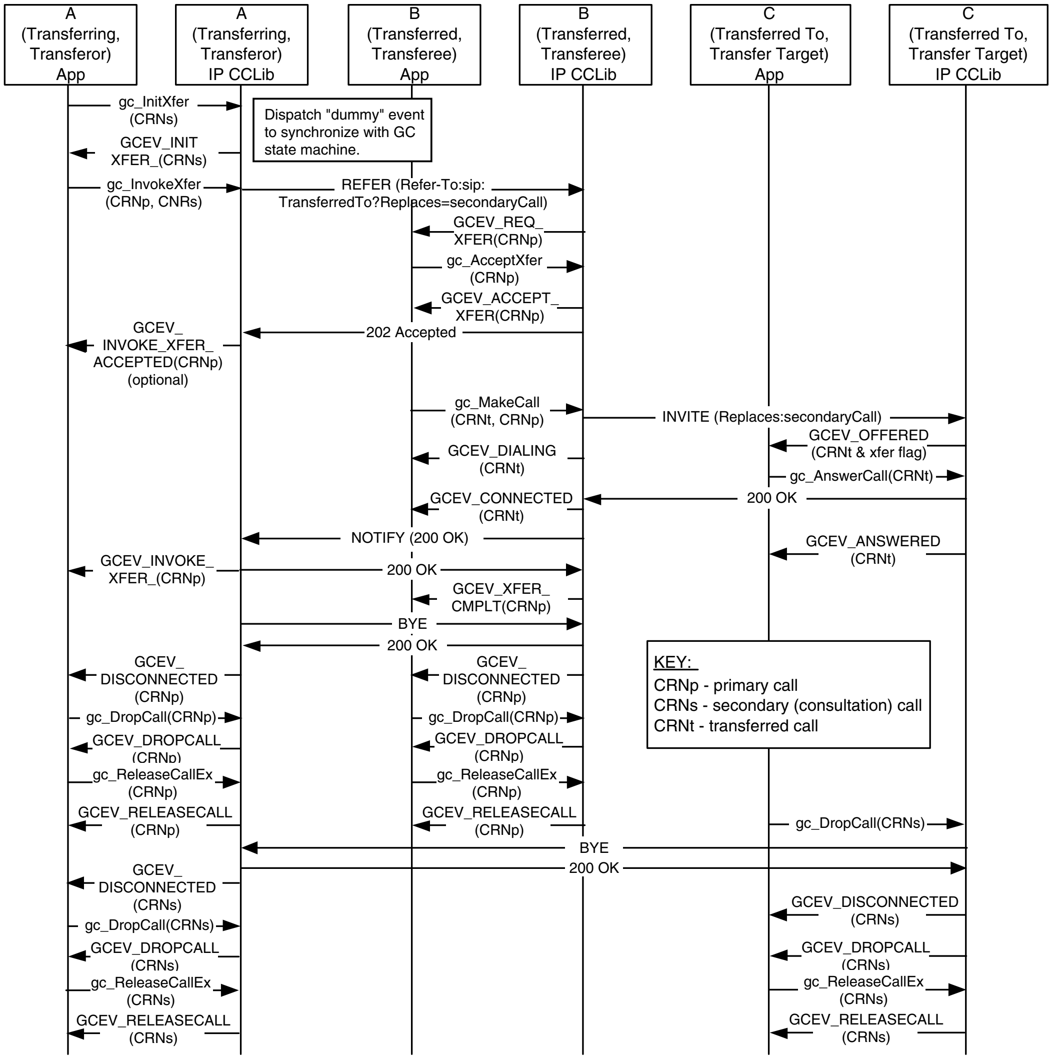
For simplification purposes, none of the figures indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related SIP calls.

3.3.5.1 Successful SIP Attended Call Transfer

Figure 25 illustrates the basic scenario for successful SIP attended call transfer. The scenario illustrates the use of a **gc_InitXfer( )** function call, which is not required in SIP. The GCEV_INIT_XFER completion event in this case contains a dummy rerouting address.

Figure 25. Successful SIP Attended Call Transfer

Pre condition: Primary call between A and B is connected (not shown).
Secondary (consultation) call between A and C is connected (not shown).



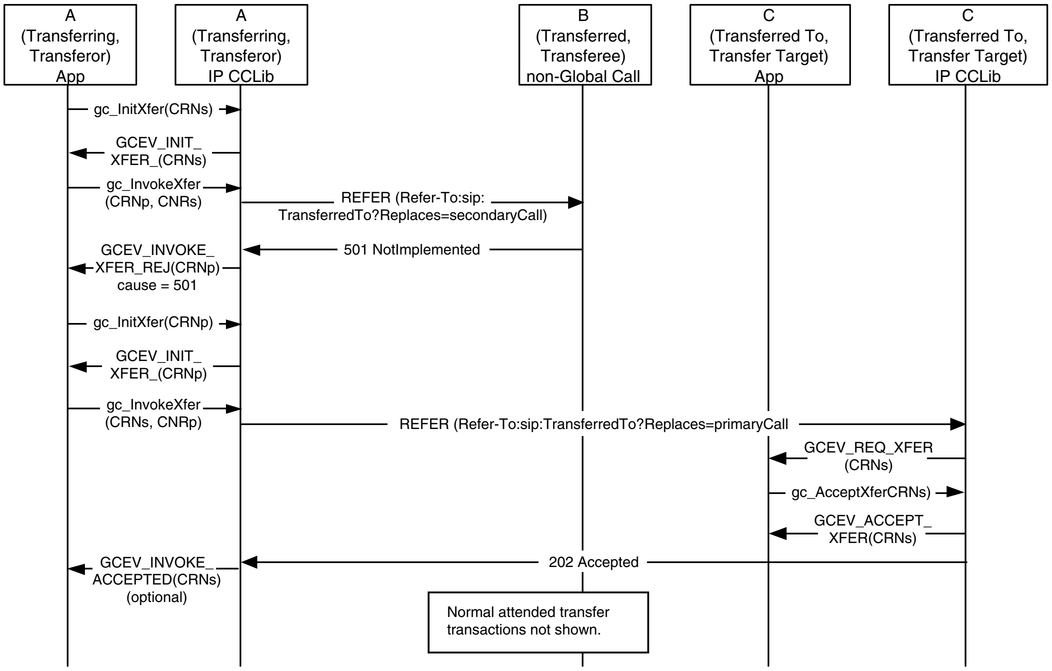
Post condition: Transferred call between B and C offered (option whether connected or not).
Primary call between A and B dropped and released.
Secondary (consultation) call between A and C dropped and released.

3.3.5.2 Attended Transfer when REFER is Not Globally Supported

If protecting or exposing the Transfer Target is not a concern, it is possible to complete a attended transfer when only the Transferor and one other party support REFER. Note that a 405 Method Not Allowed might be returned instead of the 501 Not Implemented response.

Figure 26. SIP Attended Call Transfer, Recovery from REFER Unsupported

Pre condition: Primary call between A and B is connected (not shown).
Secondary (consultation) call between A and C is connected (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) non-Global Call
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InitXfer(CRNs)
GCEV_INIT_ XFER_(CRNs)
gc_InvokeXfer (CRNp, CNRs)
REFER (Refer-To:sip: TransferredTo?Replaces=secondaryCall)
501 NotImplemented
GCEV_INVOKE_ XFER_REJ(CRNp) cause = 501
gc_InitXfer(CRNp)
GCEV_INIT_ XFER_(CRNp)
gc_InvokeXfer (CRNs, CNRp)
REFER (Refer-To:sip:TransferredTo?Replaces=primaryCall
GCEV_REQ_XFER (CRNs)
gc_AcceptXferCRNs)
GCEV_ACCEPT_ XFER(CRNs)
202 Accepted
GCEV_INVOKE_ ACCEPTED(CRNs) (optional)
Normal attended transfer transactions not shown.

Post condition: Transferred call between B and C offered (option whether connected or not).
Primary call between A and B dropped and released.
Secondary (consultation) call between A and C dropped and released.

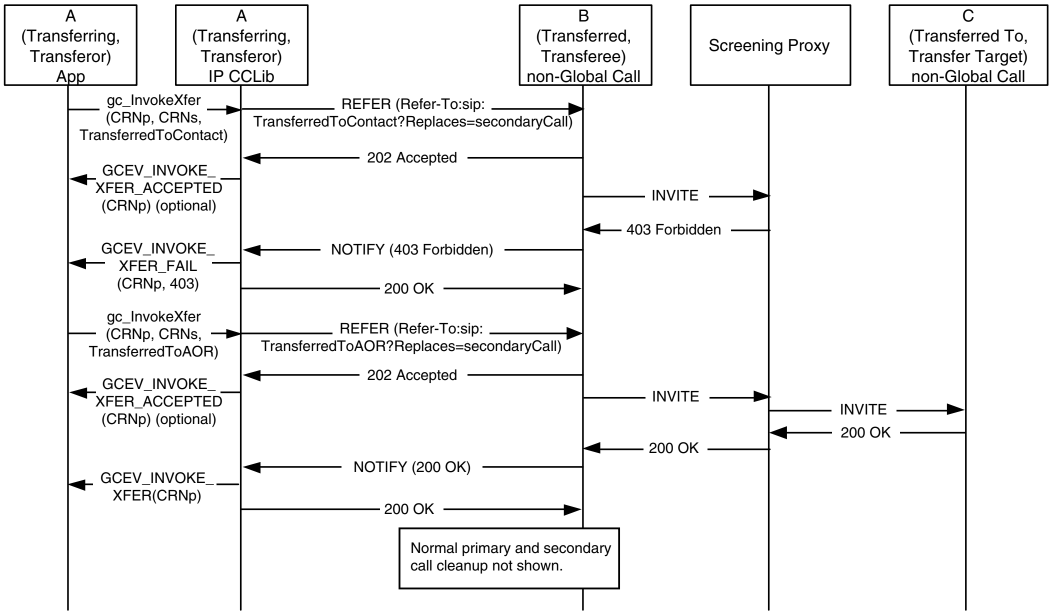
3.3.5.3 Attended Transfer When Contact URI is Not Globally Routable

It is a requirement of RFC3261 that a Contact URI be globally routable even outside the dialog. However, due to RFC2543 User Agents and some architectures (NAT/firewall traversal, screening proxies, ALGs, etc.), this will not always be the case. As a result, the methods of attended transfer shown in Figure 25 and Figure 26 may fail since they use the Contact URI in the Refer-To header field. Figure 27 shows such a scenario involving a Screening Proxy in which the transfer initially fails but succeeds on a second try. The failure response (403 Forbidden, 404 Not Found, or a timeout after no response) is communicated back to the Transferor. Since this may be caused by routing problems with the Contact URI, the Transferor retries the REFER, this time with Refer-To containing the Address of Record (AOR) of the Target (the same URI the Transferor used to reach the Transfer Target). However, the use of the AOR URI may result in routing features being activated such as forking or sequential searching which may result in the triggered INVITE reaching the wrong User Agent. To prevent an incorrect UA answering the INVITE, a Require: replaces header field is included in the Refer-To. This ensures that only the UA which matches the Replaces dialog will answer the INVITE, since any incorrect UA which supports Replaces will reply with a 481 and a UA which does not support Replaces will reply with a 420.

Note that there is still no guarantee that the correct endpoint will be reached, and the result of this second REFER may also be a failure. In that case, the Transferor could fall back to unattended transfer or give up on the transfer entirely. Since two REFERs are sent within the dialog, creating two distinct subscriptions, the Transferee uses the 'id' parameter in the Event header field to distinguish notifications for the two subscriptions.

Figure 27. SIP Attended Call Transfer, Recovery from URI Not Routable

Pre condition: Primary call between A and B is connected (not shown).
Secondary (consultation) call between A and C is connected (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) non-Global Call
Screening Proxy
C (Transferred To, Transfer Target) non-Global Call
gc_InvokeXfer (CRNp, CRNs, TransferredToContact)
REFER (Refer-To:sip: TransferredToContact?Replaces=secondaryCall)
202 Accepted
GCEV_INVOKE_ XFER_ACCEPTED (CRNp) (optional)
INVITE
403 Forbidden
GCEV_INVOKE_ XFER_FAIL (CRNp, 403)
NOTIFY (403 Forbidden)
200 OK
gc_InvokeXfer (CRNp, CRNs, TransferredToAOR)
REFER (Refer-To:sip: TransferredToAOR?Replaces=secondaryCall)
202 Accepted
GCEV_INVOKE_ XFER_ACCEPTED (CRNp) (optional)
INVITE
INVITE
200 OK
200 OK
NOTIFY (200 OK)
GCEV_INVOKE_ XFER(CRNp)
200 OK
Normal primary and secondary call cleanup not shown.

Post condition: Transferred call between B and C is connected.
Primary and secondary calls are dropped and released.

3.3.6 Unsuccessful Call Transfer Scenarios

All of the scenarios in this section apply to both unattended (blind) transfer and attended (supervised) SIP call transfers. The **gc_InitXfer( )** function call and GCEV_INIT_XFER termination event are “dummy” operations that are only used to synchronize the Global Call state machine and can safely be ignored in this context.

Transfer failures can be caused by any of transfer endpoints as shown in scenarios. In all cases, the transferring endpoint (Transferor or party A) is notified by either INVOKE_XFER_REJ or INVOKE_XFER_FAIL event with cause. No NOTIFY will be sent from party B to party A if REFER is not accepted by 202 Accepted from party B. The primary call and secondary call, if any, remain in connected state after any transfer failure.

The most common transfer failure scenarios are described in the following topics:

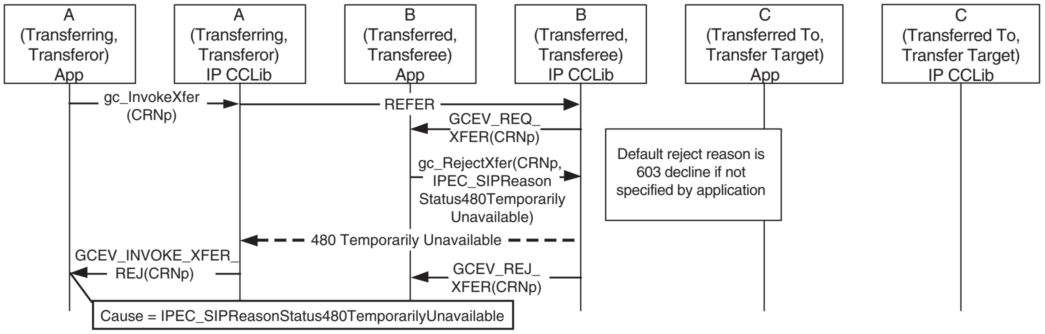
- Party B Rejects Call Transfer
- No Response From Party B
- No Response From Party C
- Party B Drops Transferred Call Early
- Party C is Busy When Transfer Attempted

3.3.6.1 Party B Rejects Call Transfer

Figure 28, illustrates a scenario in which the application at the transferred endpoint (Transferee or party B) calls **gc_RejectXfer( )** to signal the Transferor (party A) that it cannot participate in a transfer. The application may specify any valid SIP rejection reason, such as the 480 Temporarily Unavailable shown in the figure; if no reason is specified, the default reason sent is 603 Decline. As a result of the rejection, the GCEV_INVOKE_XFER_REJ termination event is received at the Transferor application (party A). The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

Figure 28. SIP Call Transfer Failure - Party B Rejects Call Transfer

Pre condition: Primary call between A and B is connected (not shown).

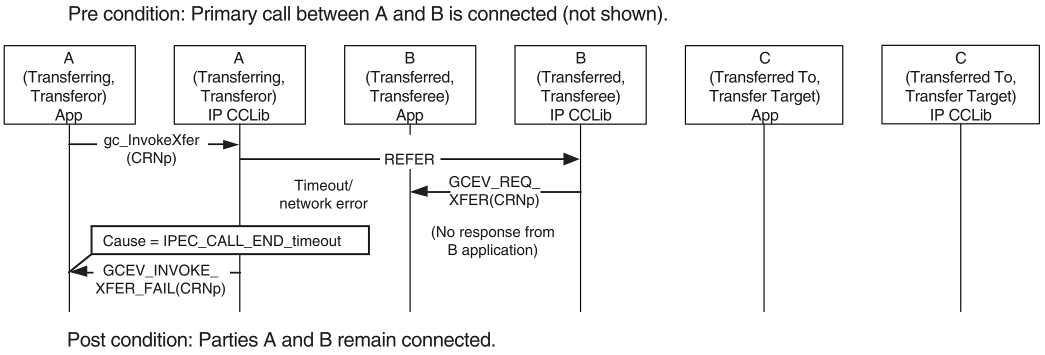

A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_ XFER(CRNp)
gc_RejectXfer(CRNp, IPEC_SIPReason Status480Temporarily Unavailable)
Default reject reason is 603 decline if not specified by application
480 Temporarily Unavailable
GCEV_INVOKE_XFER_ REJ(CRNp)
GCEV_REJ_ XFER(CRNp)
Cause = IPEC_SIPReasonStatus480TemporarilyUnavailable

Post condition: Parties A and B remain connected.

3.3.6.2 No Response From Party B

Figure 29 a scenario in which the Transferee (party B) does not respond to the REFER, causing the T3 timer at the party A (configured as 20 seconds) to expire. After the timeout, the Transferor application receives the GCEV_INVOKE_XFER_FAIL termination event. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

Figure 29. SIP Call Transfer Failure - No Response from Party B

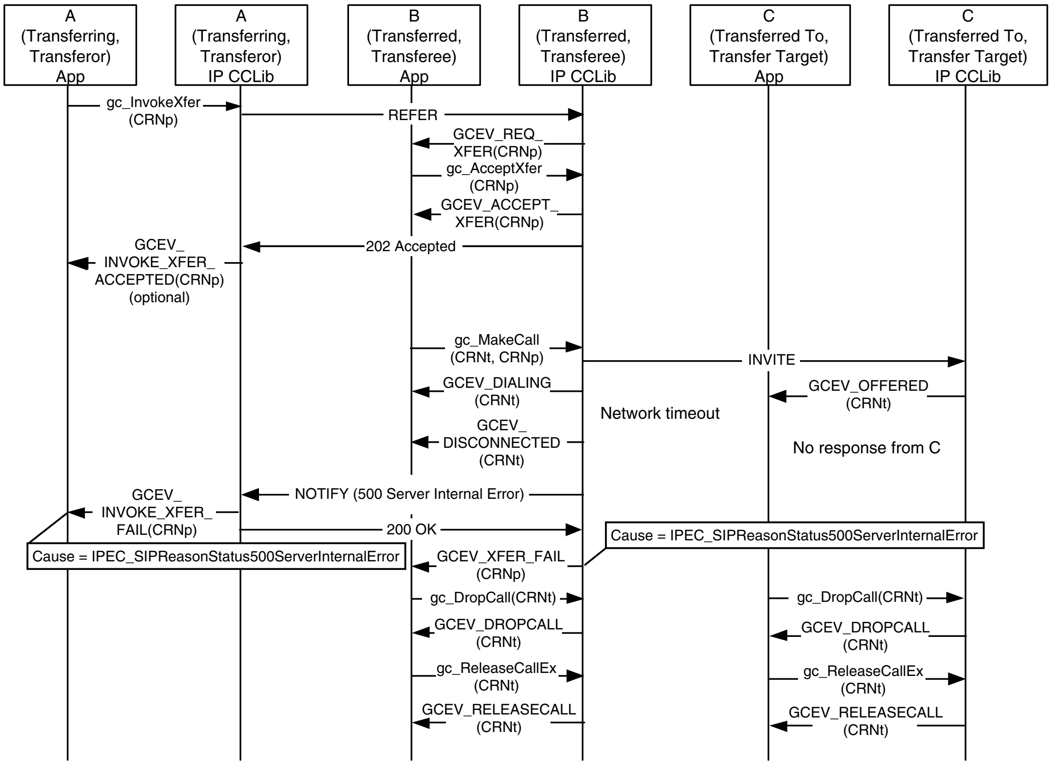

Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
Timeout/ network error
GCEV_REQ_ XFER(CRNp)
(No response from B application)
Cause = IPEC_CALL_END_timeout
GCEV_INVOKE_ XFER_FAIL(CRNp)
Post condition: Parties A and B remain connected.

3.3.6.3 No Response From Party C

Figure 30 illustrates a scenario in which the Transfer Target (party C) does not respond to the incoming call from the Transferee (party B) which causes the T4 timer at party B (configured as 20 seconds) to expire. As a result, the Transferee application (party B) receives the GCEV_DISCONNECT event for the transferred call timeout. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 30. SIP Call Transfer Failure - No Response from Party C

Pre condition: Primary call between A and B is connected (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_XFER(CRNp)
gc_AcceptXfer (CRNp)
GCEV_ACCEPT_XFER(CRNp)
202 Accepted
GCEV_INVOKE_XFER_ACCEPTED(CRNp) (optional)
gc_MakeCall (CRNt, CRNp)
INVITE
GCEV_DIALING (CRNt)
GCEV_OFFERED (CRNt)
Network timeout
GCEV_DISCONNECTED (CRNt)
No response from C
NOTIFY (500 Server Internal Error)
GCEV_INVOKE_XFER_FAIL(CRNp)
200 OK
Cause = IPEC_SIPReasonStatus500ServerInternalError
Cause = IPEC_SIPReasonStatus500ServerInternalError
GCEV_XFER_FAIL (CRNp)
gc_DropCall(CRNt)
gc_DropCall(CRNt)
GCEV_DROPCALL (CRNt)
GCEV_DROPCALL (CRNt)
gc_ReleaseCallEx (CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_RELEASECALL (CRNt)
GCEV_RELEASECALL (CRNt)

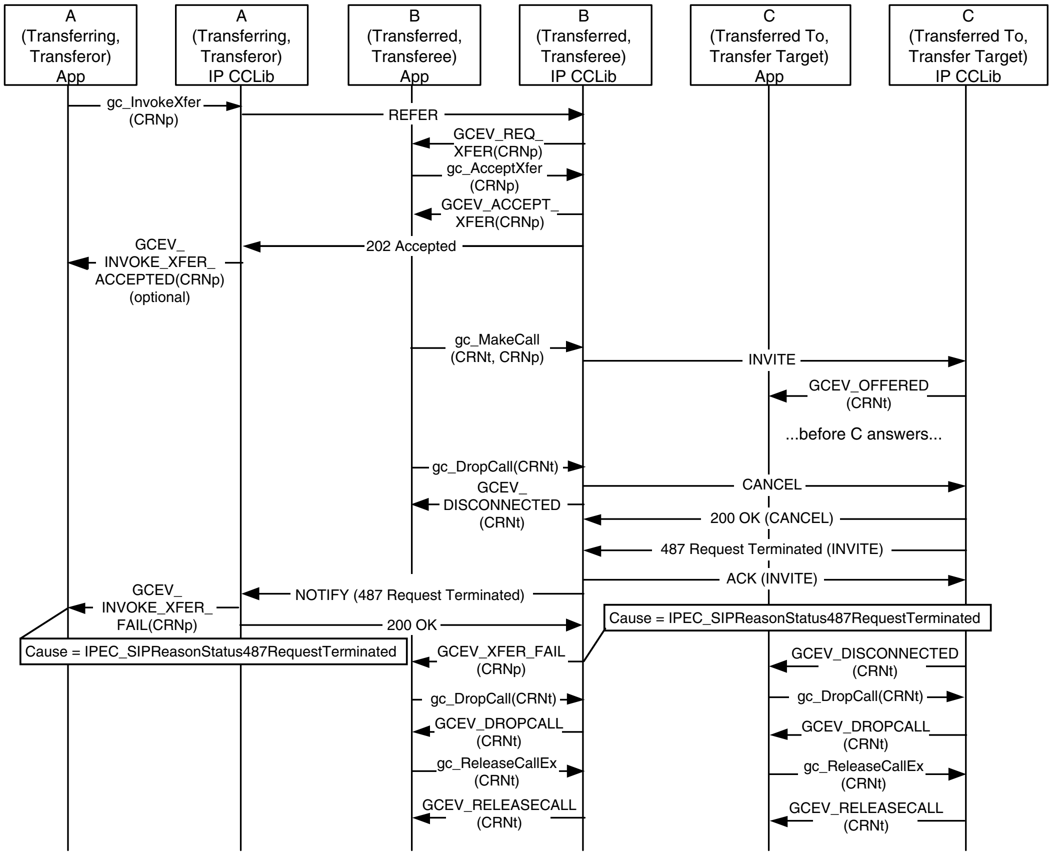
Post condition: Parties A and B remain connected.

3.3.6.4 Party B Drops Transferred Call Early

Figure 31 illustrates a scenario in which the Transferee (party B) drops the transferred call before receiving a response to the INVITE it sent to party C. As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at the Transferor (party A) and the GCEV_XFER_FAIL termination event is received a the Transferee (party B). The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 31. SIP Call Transfer Failure - Party B Drops Transferred Call Early

Pre condition: Primary call between A and B is connected (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_ XFER(CRNp)
gc_AcceptXfer (CRNp)
GCEV_ACCEPT_ XFER(CRNp)
202 Accepted
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp) (optional)
gc_MakeCall (CRNt, CRNp)
INVITE
GCEV_OFFERED (CRNt)
...before C answers...
gc_DropCall(CRNt)
CANCEL
GCEV_ DISCONNECTED (CRNt)
200 OK (CANCEL)
487 Request Terminated (INVITE)
ACK (INVITE)
GCEV_ INVOKE_XFER_ FAIL(CRNp)
NOTIFY (487 Request Terminated)
200 OK
Cause = IPEC_SIPReasonStatus487RequestTerminated
Cause = IPEC_SIPReasonStatus487RequestTerminated
GCEV_XFER_FAIL (CRNp)
GCEV_DISCONNECTED (CRNt)
gc_DropCall(CRNt)
gc_DropCall(CRNt)
GCEV_DROPCALL (CRNt)
GCEV_DROPCALL (CRNt)
gc_ReleaseCallEx (CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_RELEASECALL (CRNt)
GCEV_RELEASECALL (CRNt)

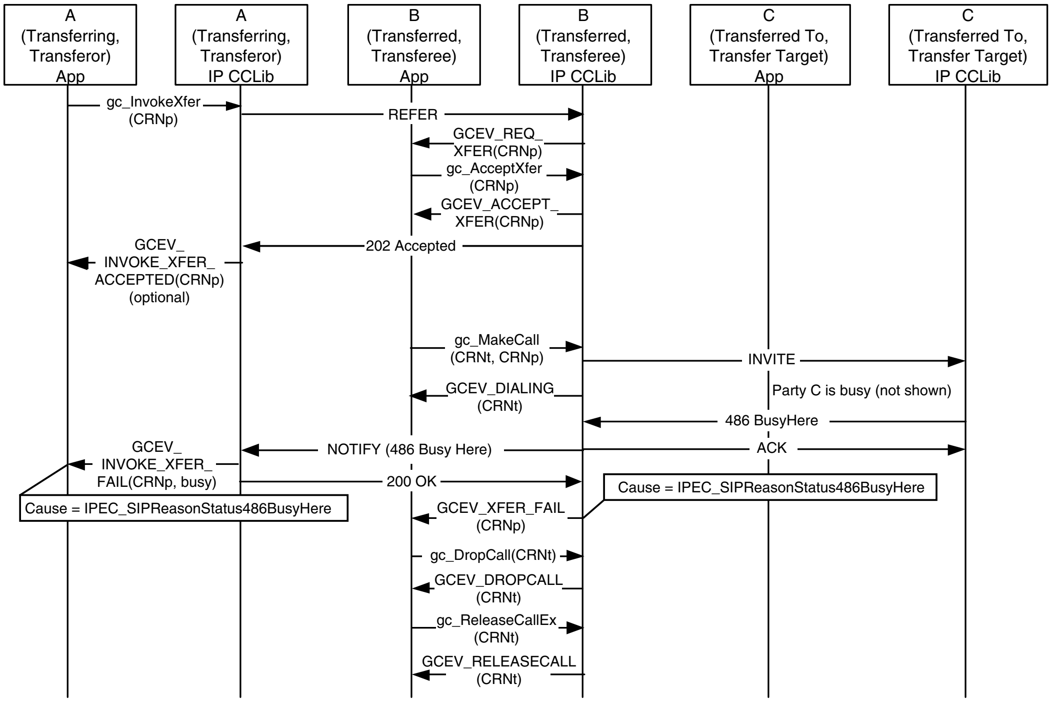
Post condition: Parties A and B remain connected.

3.3.6.5 Party C is Busy When Transfer Attempted

Figure 32 illustrates a scenario in which the Transfer Target (party C) is busy at the time the transfer is requested. (This primarily applies to unattended transfers, since the Transferor would be aware that the Transfer Target is busy in an attended transfer.) In this case, the Transferor (party A) receives a GCEV_INVOKE_XFER_FAIL termination event and the Transferee (party B) receives a GCEV_XFER_FAIL termination event. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

Figure 32. SIP Call Transfer Failure - Party C is Busy When Transfer Attempted

Pre condition: Primary call between parties A and B is connected (not shown).
Party C has call connected to another party (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_ XFER(CRNp)
gc_AcceptXfer (CRNp)
GCEV_ACCEPT_ XFER(CRNp)
202 Accepted
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp) (optional)
gc_MakeCall (CRNt, CRNp)
INVITE
GCEV_DIALING (CRNt)
Party C is busy (not shown)
486 BusyHere
GCEV_ INVOKE_XFER_ FAIL(CRNp, busy)
NOTIFY (486 Busy Here)
ACK
200 OK
Cause = IPEC_SIPReasonStatus486BusyHere
Cause = IPEC_SIPReasonStatus486BusyHere
GCEV_XFER_FAIL (CRNp)
gc_DropCall(CRNt)
GCEV_DROPCALL (CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_RELEASECALL (CRNt)

Post condition: Parties A and B remain connected.
Party C also remains connected (to another party not shown).

3.4 T.38 Fax Server Call Scenarios

Global Call supports T.38 fax server as described in Section 4.26, "T.38 Fax Server", on page 247. The following scenarios demonstrate the T.38 fax server capabilities provided when using IP technology (both H.323 and SIP):

- Sending T.38 Fax in an Established Audio Session
- Receiving T.38 Fax in an Established Audio Session
- Sending T.38 Fax Without an Established Audio Session
- Receiving T.38 Fax Without an Established Audio Session
- Sending a Request to Switch From T.38 Fax to Audio
- Receiving a Request to Switch From T.38 Fax to Audio
- Terminating a Call After a T.38 Fax Session
- Recovering from a Session Switching Failure

Note: In these scenarios, the application must include T.38 Fax capability either when using **gc_MakeCall( )** for an outbound call or when using **gc_CallAck( )**, **gc_AcceptCall( )**, or **gc_AnswerCall( )** for an inbound call.

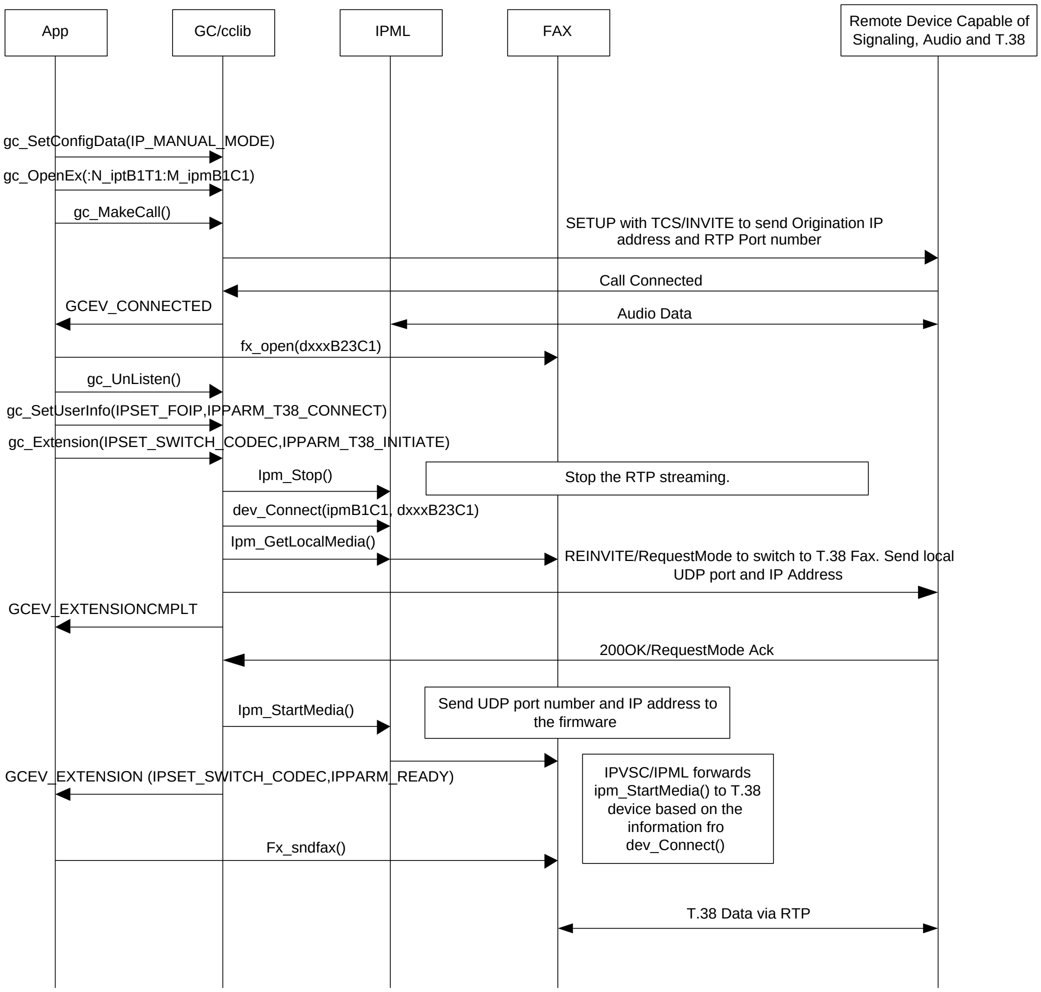
3.4.1 Sending T.38 Fax in an Established Audio Session

In this scenario, the user application uses the Global Call API to open a Media device, configures "Manual" mode of operation and establishes an audio session with the remote device. See Section 4.26.2, "Specifying Manual Operating Mode", on page 248 for more information on manual mode. A T.38 Fax device is then opened and the application switches from an audio session to a T.38 session.

When the application receives notification that the T.38 session is ready, fax information can be sent. Figure 33 shows the scenario diagram.

Note: The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 33 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 33. Sending T.38 Fax in an Established Audio Session


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
SETUP with TCS/INVITE to send Origination IP address and RTP Port number
Call Connected
GCEV_CONNECTED
Audio Data
fx_open(dxxxB23C1)
gc_UnListen()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_T38_INITIATE)
Ipm_Stop()
Stop the RTP streaming.
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
REINVITE/RequestMode to switch to T.38 Fax. Send local UDP port and IP Address
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
Ipm_StartMedia()
Send UDP port number and IP address to the firmware
GCEV_EXTENSION (IPSET_SWITCH_CODEC,IPPARM_READY)
IPVSC/IPML forwards ipm_StartMedia() to T.38 device based on the information fro dev_Connect()
Fx_sndfax()
T.38 Data via RTP

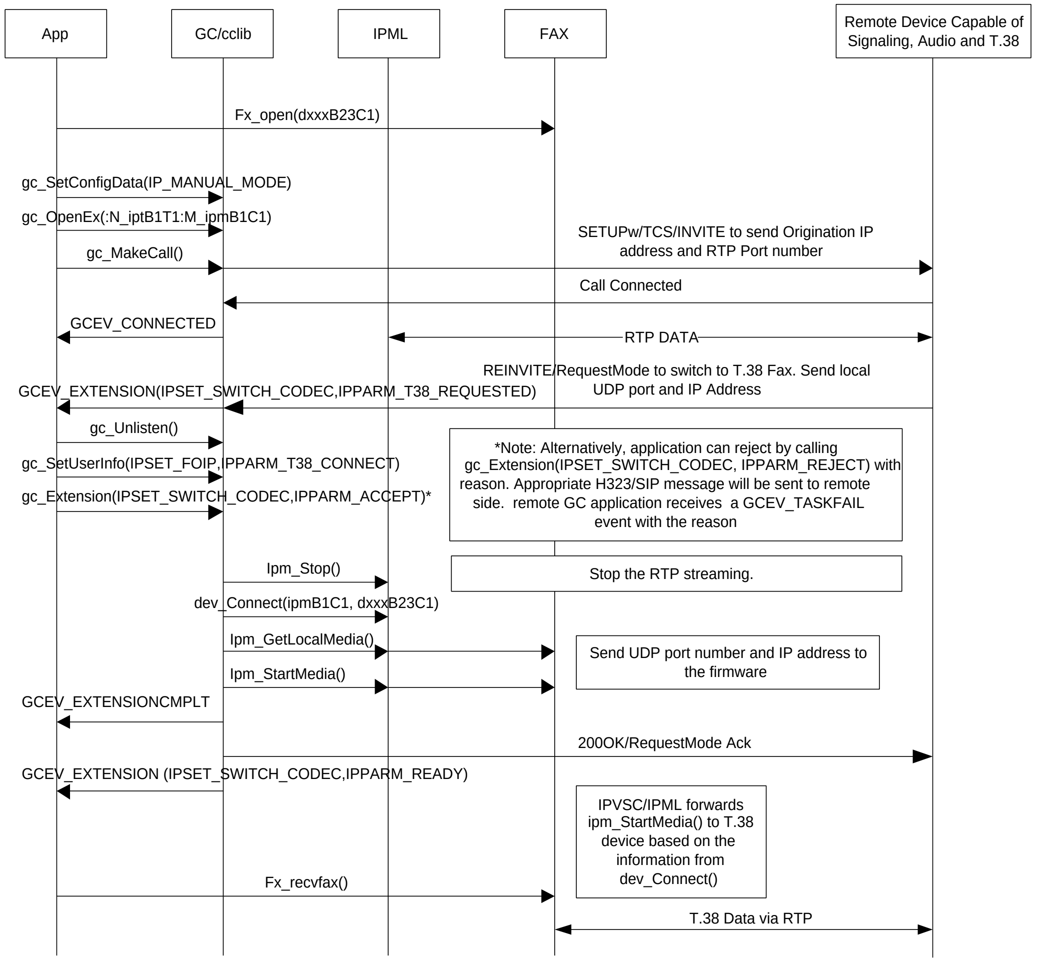
3.4.2 Receiving T.38 Fax in an Established Audio Session

In this scenario, the user application uses the Global Call API to open a Media device, configures "Manual" operating mode and establishes an audio session with the remote device. See Section 4.26.2, "Specifying Manual Operating Mode", on page 248 for more information on manual mode. To prepare to receive fax, the application must also open a T.30 Fax device. During the audio session, the application can be notified of an incoming request to switch from audio to T.38 fax.

The application can choose to accept or reject this request. If the user chooses to accept, Global Call notifies the application that the T.38 session is ready to receive a fax. Figure 34 shows the scenario diagram.

Note: The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 34 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 34. Receiving T.38 Fax in an Established Audio Session


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
Fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
SETUPw/TCS/INVITE to send Origination IP address and RTP Port number
gc_MakeCall()
Call Connected
GCEV_CONNECTED
RTP DATA
REINVITE/RequestMode to switch to T.38 Fax. Send local UDP port and IP Address
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_T38_REQUESTED)
gc_Unlisten()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_ACCEPT)*
*Note: Alternatively, application can reject by calling gc_Extension(IPSET_SWITCH_CODEC, IPPARM_REJECT) with reason. Appropriate H323/SIP message will be sent to remote side. remote GC application receives a GCEV_TASKFAIL event with the reason
Ipm_Stop()
Stop the RTP streaming.
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
Send UDP port number and IP address to the firmware
Ipm_StartMedia()
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
GCEV_EXTENSION (IPSET_SWITCH_CODEC,IPPARM_READY)
IPVSC/IPML forwards ipm_StartMedia() to T.38 device based on the information from dev_Connect()
Fx_recvfax()
T.38 Data via RTP

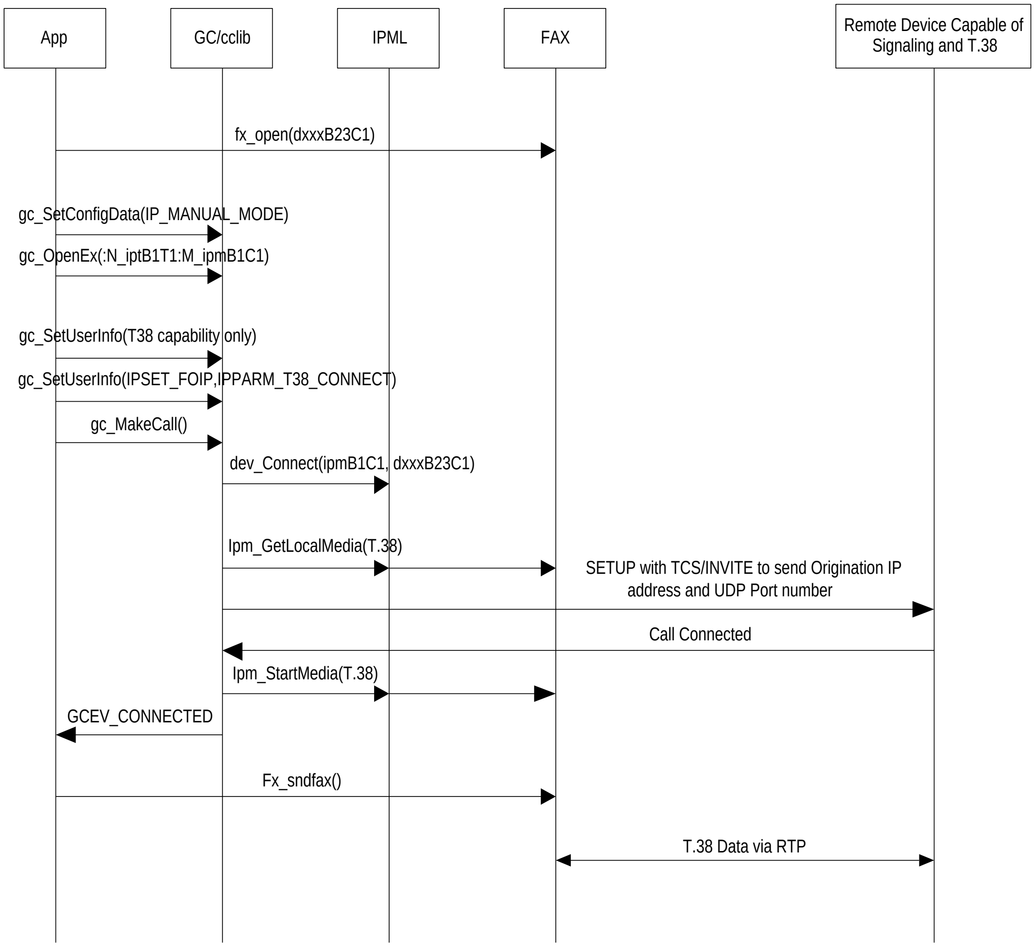
3.4.3 Sending T.38 Fax Without an Established Audio Session

This scenario describes the sending of T.38 Fax in a media session that does not have audio already established. The application first opens a Media device and a T.38 Fax device and configures "Manual" mode of operation. See Section 4.26.2, "Specifying Manual Operating Mode", on page 248 for more information on manual mode. The Global Call API is then used to associate the T.38 Fax device with the IP Media device before making a new T.38 call.

Once the call is connected, the application can send a fax. Figure 35 shows the scenario diagram.

Note: The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 35 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 35. Sending T.38 Fax Without an Established Audio Session


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling and T.38
fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_SetUserInfo(T38 capability only)
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_MakeCall()
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia(T.38)
SETUP with TCS/INVITE to send Origination IP address and UDP Port number
Call Connected
Ipm_StartMedia(T.38)
GCEV_CONNECTED
Fx_sndfax()
T.38 Data via RTP

3.4.4 Receiving T.38 Fax Without an Established Audio Session

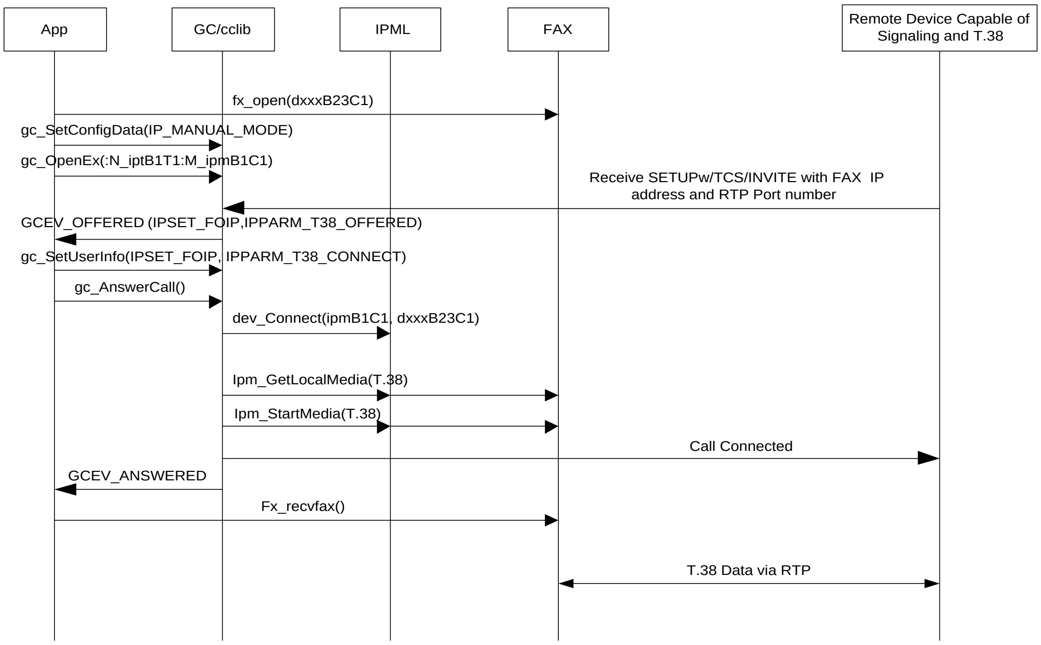
This scenario describes the reception of T.38 Fax in a media session that does not have audio already established. The application first opens a Media device and a T.38 Fax device and configures “Manual” operating mode. See Section 4.26.2, “Specifying Manual Operating Mode”, on page 248 for more information on manual mode. When the application receives a T.38 fax request, a GCEV_OFFERED event with T.38 extension information is received.

If the application accepts the call, the T.38 Fax device is associated with the Media device before the T.38 call is answered. Figure 36 shows the scenario diagram.

Notes: **1.** The GCEV_OFFERED event with T.38 extension information is only generated if the following requirements are met. For H.323, the incoming message must be a Q.931 Setup message with data terminal capability only. For SIP, the incoming message must be an INVITE message with an SDP that has an image media descriptor only. If this condition is not met, the GCEV_OFFERED event does not include any T.38 extension information. This limitation prevents the T.38 server from receiving the T.38 request in H.323 slow start or in a SIP no SDP INVITE request.

2. The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 36 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 36. Receiving T.38 Fax Without an Established Audio Session

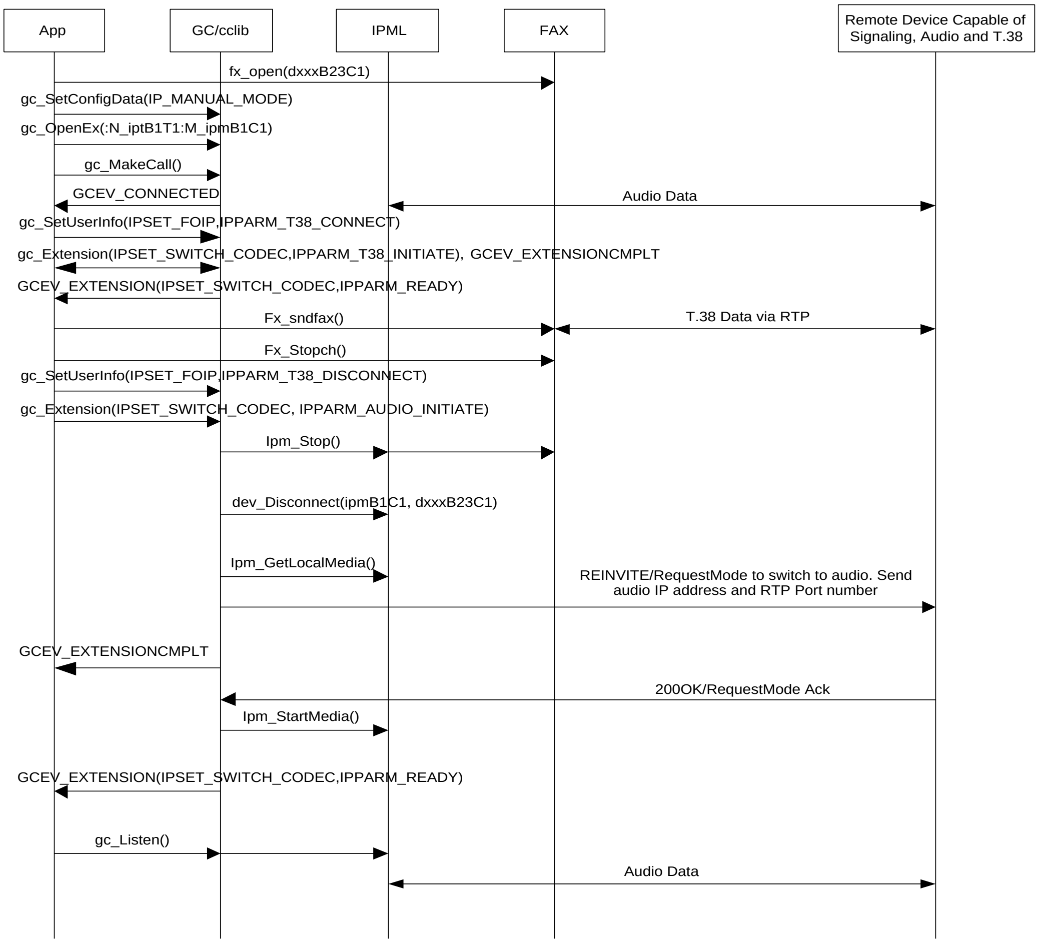

App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling and T.38
fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
Receive SETUPw/TCS/INVITE with FAX IP address and RTP Port number
GCEV_OFFERED (IPSET_FOIP,IPPARM_T38_OFFERED)
gc_SetUserInfo(IPSET_FOIP, IPPARM_T38_CONNECT)
gc_AnswerCall()
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia(T.38)
Ipm_StartMedia(T.38)
Call Connected
GCEV_ANSWERED
Fx_recvfax()
T.38 Data via RTP

3.4.5 Sending a Request to Switch From T.38 Fax to Audio

In a fax session, when a fax completes, the application can use the Global Call API to issue a request to switch from a T.38 fax session back to an audio session after disassociating the T.38 Fax device from the Media device. When Global Call notifies the application that the audio session has been reestablished, the application can once again send and receive audio. Figure 37 shows the scenario diagram.

Note: The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 37 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 37. Sending a Request to Switch From T.38 Fax to Audio

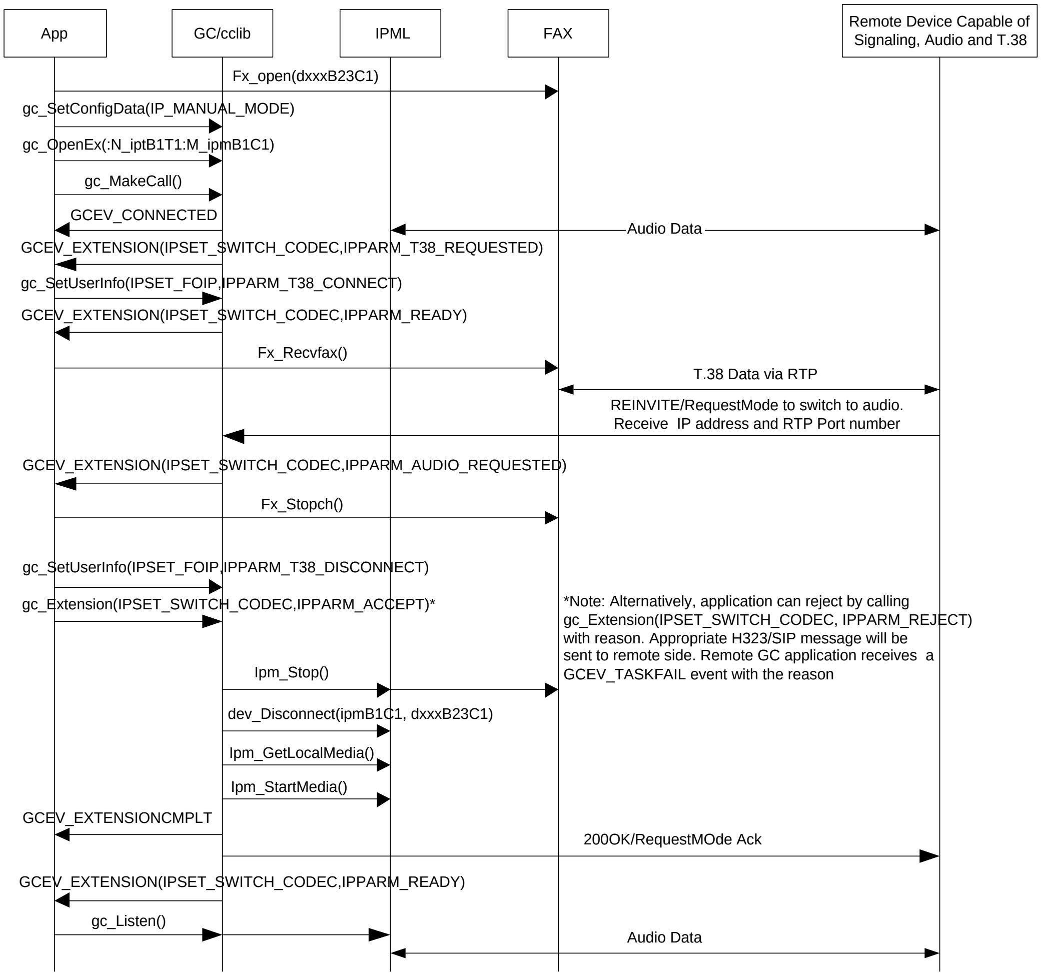

App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
GCEV_CONNECTED
Audio Data
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_T38_INITIATE), GCEV_EXTENSIONCMPLT
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
Fx_sndfax()
T.38 Data via RTP
Fx_Stopch()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_DISCONNECT)
gc_Extension(IPSET_SWITCH_CODEC, IPPARM_AUDIO_INITIATE)
Ipm_Stop()
dev_Disconnect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
REINVITE/RequestMode to switch to audio. Send audio IP address and RTP Port number
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
Ipm_StartMedia()
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
gc_Listen()
Audio Data

3.4.6 Receiving a Request to Switch From T.38 Fax to Audio

In a fax session, when a fax completes, the application can receive a request to switch from a T.38 fax session back to an audio session. The application can choose to accept the request after disassociating the T.38 Fax device from the Media device. When Global Call notifies the application that the audio session has been reestablished, the application can once again send and receive audio. Figure 38 shows the scenario diagram.

Note: The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 38 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 38. Receiving a Request to Switch From T.38 Fax to Audio


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
Fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
GCEV_CONNECTED
Audio Data
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_T38_REQUESTED)
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
Fx_Recvfax()
T.38 Data via RTP
REINVITE/RequestMode to switch to audio.
Receive IP address and RTP Port number
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_AUDIO_REQUESTED)
Fx_Stopch()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_DISCONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_ACCEPT)*
*Note: Alternatively, application can reject by calling gc_Extension(IPSET_SWITCH_CODEC, IPPARM_REJECT) with reason. Appropriate H323/SIP message will be sent to remote side. Remote GC application receives a GCEV_TASKFAIL event with the reason
Ipm_Stop()
dev_Disconnect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
Ipm_StartMedia()
GCEV_EXTENSIONCMPLT
200OK/RequestMOde Ack
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
gc_Listen()
Audio Data

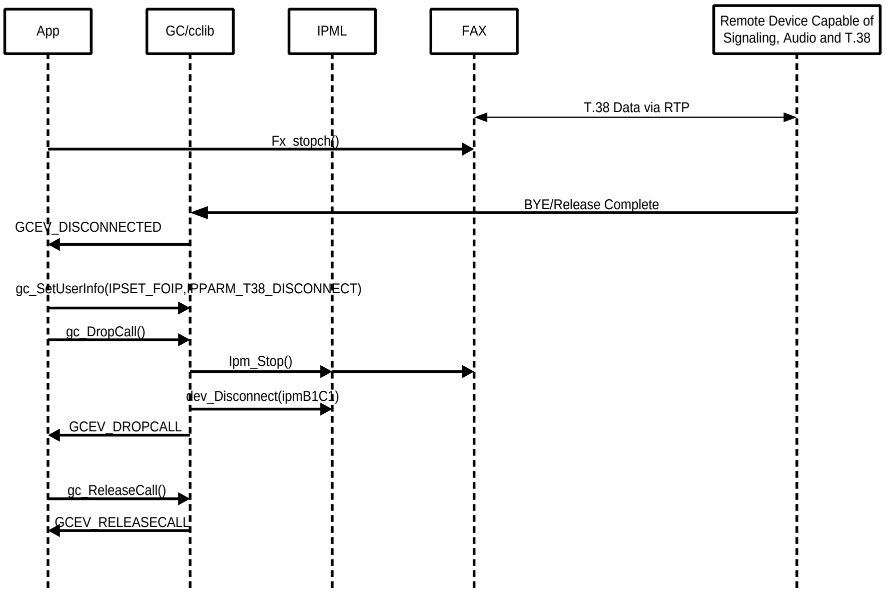
3.4.7 Terminating a Call After a T.38 Fax Session

In any scenario where a T.38 session is established and fax is complete, the application can terminate the call without switching to audio. In either outbound or inbound call termination, the application must disassociate the T.38 Fax device from the Media device before calling **gc_DropCall( )**. This ensures the Media device in the correct state for the next call.

Terminating a call after an audio session follows the normal Global Call call procedures.

Note: The application must not use both Global Call and IP Media Library functions on the same device. The IP Media Library calls (ipm_) in Figure 39 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 39. Terminating a Call After a T.38 Fax Transfer.



3.4.8 Recovering from a Session Switching Failure

Switching to T.38 Fax or audio may fail due to any a number of reasons, for example, rejection or no response from remote endpoint. It is highly recommended that the application set up a timer for a minimum of 35 seconds for each switching request. If a timeout occurs while waiting for a GCEV_EXTENSION event that has an associated IPPARM_READY parameter, the application has two options:

- Attempt to switch back to original session as if the GCEV_EXTENSION event were received without media capability.
- Terminate the call as if GCEV_EXTENSION event were received without media capability.

If the application times out when switching to T.38 Fax (that is, it does not receive a GCEV_EXTENSION event with an IPPARM_READY parameter within the timeout period), it should follow the scenarios described in Section 3.4.5, “Sending a Request to Switch From T.38

Fax to Audio", on page 92, Section 3.4.6, "Receiving a Request to Switch From T.38 Fax to Audio", on page 93, or Section 3.4.7, "Terminating a Call After a T.38 Fax Session", on page 94.

Note: The application must call the **gc_SetUserInfo( )** function with a GC_PARM_BLK that contains a set ID of IPSET_FOIP and a parameter ID of IPPARM_T38_DISCONNECT to disassociate the devices in any of the scenarios.

If the application times out when switching to audio (that is, it does not receive a GCEV_EXTENSION event with an IPPARM_READY parameter within the timeout period), it should follow the scenarios described in Section 3.4.1, "Sending T.38 Fax in an Established Audio Session", on page 88, Section 3.4.2, "Receiving T.38 Fax in an Established Audio Session", on page 89, or drop the call as if in audio session.

IP-Specific Operations 4

This chapter describes how to use Global Call to perform certain operations in an IP environment. These operations include:

4.1 Call Control Library Initialization

Certain configuration parameters, such as the maximum number of IPT devices available, the local IP address, the call signaling port, and the outbound proxy for SIP, are configurable when using the **gc_Start( )** function to initialize Global Call. For example, the default maximum number of IPT devices is 120, but is configurable to as many as 2016. Other items that may be configured by the application are the terminal type, the call transfer supplementary service, the ability to access H.323 message information fields, and the ability to access SIP message header fields and MIME-encoded message bodies.

Before using the **gc_Start( )** function, the application sets the desired configuration in a IPCCLIB_START_DATA data structure, and an array of IP_VIRTBOARD data structures (one per virtual board) that are referenced by it. Before setting any of the configuration items, the application must use the **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions to initialize the IPCCLIB_START_DATA structure and each IP_VIRTBOARD data structures. These functions set default values that can then be overridden with desired values. After setting whatever non-default values it desires (there is no need for the application to set any item that it is leaving at the default value), the application references the IPCCLIB_START_DATA structure from a CCLIB_START_STRUCT structure, which in turn is referenced from the GC_START_STRUCT structure that is passed to the **gc_Start( )** function.

See Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340, the reference page for IP_VIRTBOARD on page 394, and the reference page for IPCCLIB_START_DATA on page 398 for more information on the defaut values and the allowable values that the application can set for each configuration item.

Note: In the IPCCLIB_START_DATA structure, the maximum value of the num_boards field, which defines the number of NICs or NIC addresses, is 8.

4.1.1 Setting a SIP Outbound Proxy

When initializing a board device for use with SIP, the application can set an outbound proxy. When such a proxy is set, all outbound requests are sent to the proxy address rather than the IP address of the original Request-URI. The proxy can be set by specifying an IP address or a host name in the IP_VIRTBOARD structure that is used in the **gc_Start( )** function. If both an IP address and a host name are specified in IP_VIRTBOARD, the IP address takes precedence.

The following code snippet illustrates how to set a SIP outbound proxy for a single board:

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);
```

```
/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// set outbound proxy by IP Address
virtBoards[0].outbound_proxy_IP.ip_ver = IPVER4;
virtBoards[0].outbound_proxy_IP.u_ipaddr.ipv4 = inet_addr("192.168.1.227");

// set outbound proxy by hostname.
// if outbound proxy is also set by IP address, this is ignored
char OutboundProxyHostName[256];
strcpy(OutboundProxyHostName,"my_outbound_proxy");
virtBoard[0].outbound_proxy_hostname = OutboundProxyHostName;

// set outbound proxy port
virtBoards[0].outbound_proxy_port = 5060;
```

4.1.2 Configuring SIP Transport Protocol

When initializing a board device for use with SIP, the application can enable the use of the TCP transport protocol in addition to the default UDP transport.

When TCP is enabled, the Global Call library listens for incoming TCP connections as well as UDP connections on the SIP signaling port that is configured for the board.

When TCP is enabled, an outbound message is sent using TCP if any of the following three conditions is true:

- The board device was configured with TCP as the default transport protocol if there is no proxy, or with TCP as the outbound proxy protocol if there is a SIP proxy configured.
- TCP is explicitly specified by setting the string “;transport=tcp” in the Request-URI header field before the message is sent. (Note that this requires the SIP Message Info feature to have been enabled by setting the IP_SIP_MSGINFO_ENABLE mask value in the sip_msginfo_mask field of IP_VIRTBOARD before starting the board.)
- The size of the outgoing message is larger than the configured maximum size for UDP messages, which is 1300 by default.

If none of these conditions is true, UDP is used as the default transport protocol.

The SIP transport protocol is configured by five fields in the IP_VIRTBOARD structure that is used in the **gc_Start( )** function:

E_SIP_tcpenabled
: Enables TCP support. The default value disables TCP so that all outgoing messages are sent over UDP and incoming TCP messages are refused. No TCP capabilities are available unless this parameter is set to the Enabled value.

E_SIP_OutboundProxyTransport
: Sets the transport protocol that is used by the SIP outbound proxy if the virtual board is configured with a proxy and TCP is enabled. The default value sets UDP as the transport for the proxy. Setting this parameter to the TCP value when TCP is not enabled, or when TCP is enabled but no proxy is configured causes a bad parameter error when **gc_Start( )** is called.

E_SIP_Persistence

Sets the persistence for TCP connections, with options for no persistence (connection closed after each request), transaction persistence (connection closed when transaction is completed), or user persistence (connection maintained for the lifetime of the user of the transaction). The default is user persistence, which minimizes the number of times that sockets are set up and torn down.

SIP_maxUDPmsgLen

Sets a maximum size for UDP messages. If TCP is enabled and the application attempts to send a message by UDP that exceeds the configured maximum size (default is 1300 as suggested in RFC3261), TCP transport is automatically used rather than UDP. This size checking may have an undesirable effect on system performance, and a parameter value is defined which disables the feature.

E_SIP_DefaultTransport

Sets the default transport protocol for requests when there is no SIP outbound proxy. The default value sets UDP as the default transport protocol. Setting this parameter to the TCP value when TCP is not enabled causes a bad parameter error when **gc_Start( )** is called. If TCP is enabled, the application can override the default transport for a specific request by explicitly setting a “transport= ” parameter in the Request-URI header field before sending the request.

See the reference page for IP_VIRTBOARD on page 394, for full details on the data structure fields and values.

4.1.2.1 Configuring TCP Transport

With five configuration items controlling TCP transport, the number of possible configuration combinations is clearly very large. The tables in this section list the combinations of configuration parameter settings that are used to achieve various system behaviors. Note that the tables include entries for the outbound proxy configuration, since the transport configuration differs depending on whether or not a proxy is enabled, and the SIP message information mask, which must be configured to allow the transport to be set for individual requests.

The following code snippet illustrates the general procedure for setting up the IP_VIRTBOARD structure to enable TCP. This specific example sets up a SIP outbound proxy, enables TCP, and sets TCP as the default transport protocol for the proxy for a single board. Note that all data structure fields that are not explicitly set are assumed to contain their default values as configured by the **INIT_IP_VIRTBOARD( )** function.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);
```

```
/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// Enable SIP Message Info to allow transport selection for individual requests
virtBoards[0].ip_sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE;

// set outbound proxy by IP Address
virtBoards[0].outbound_proxy_IP.ip_ver = IPVER4;
virtBoards[0].outbound_proxy_IP.u_ipaddr.ipv4 = inet_addr("192.168.1.227");

// set outbound proxy port
virtBoards[0].outbound_proxy_port = 5060;

//enable and configure TCP for proxy
virtBoards[0].E_SIP_tcpenabled = ENUM_Enabled;
virtBoards[0].E_SIP_OutboundProxyTransport = ENUM_TCP;
virtBoards[0].E_SIP_Persistence = ENUM_PERSISTENCE_TRANSACT_USER;
```

Transport Parameter Combinations without Proxy

All Requests UDP

Parameter	Value
E_SIP_tcpenabled	ENUM_Disabled (default)
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	not set
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	not set
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

All Requests TCP

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	ENUM_TCP
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

Selected Requests TCP

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	1300 (default)

Parameter	Value
E_SIP_DefaultTransport	ENUM_UDP (default)
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	includes IP_SIP_MSGINFO_ENABLE
transport parameter in Request-URI	set to “;transport=tcp” on selected requests

Selected Requests UDP

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	ENUM_TCP
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	includes IP_SIP_MSGINFO_ENABLE
transport parameter in Request-URI	set to “;transport=udp” on selected requests

Transport Parameter Combinations with Proxy

All Requests UDP via Proxy

Parameter	Value
E_SIP_tcpenabled	ENUM_Disabled (default)
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	not set
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	not set
outbound_proxy_* fields	IP -or- hostname set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

Requests are sent UDP to the proxy, and the proxy sends the request onward using UDP (unless the proxy resolves the destination as being TCP, based on DNS information).

All Requests TCP via Proxy

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	ENUM_TCP
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	default (not set)

Parameter	Value
E_SIP_DefaultTransport	not set
outbound_proxy_ fields	IP -or- hostname set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

Requests are sent TCP to the proxy, and the proxy sends the request onward using TCP.

Selected Requests TCP via Proxy

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	ENUM_UDP (default)
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	1300 (default)
E_SIP_DefaultTransport	not set
outbound_proxy_ fields	IP -or- hostname set
sip_msginfo_mask	includes IP_SIP_MSGINFO_ENABLE
transport parameter in Request-URI	set to ";transport=tcp" for selected requests

Selected requests are sent TCP to the proxy, and the proxy sends the request onward using TCP. Other requests are sent UDP to proxy, and are sent onward using UDP (unless the proxy resolves the destination as being TCP, based on DNS information).

Invalid Transport Parameter Combinations

If **TCP is not enabled** (E_SIP_tcpenabled is the default ENUM_Disabled value), the following parameter settings are invalid:

- If E_SIP_OutboundProxyTransport is set to ENUM_TCP, **gc_Start( )** returns an IPERR_BAD_PARM error.
- If E_SIP_DefaultTransport is set to ENUM_TCP, **gc_Start( )** returns an IPERR_BAD_PARM error.
- Setting the Request-URI transport parameter to ";transport=tcp" is invalid but does not produce an error. The invalid header field parameter is ignored, and the request is sent using UDP.

If **TCP is enabled** (E_SIP_tcpenabled is set to ENUM_Enabled), and **no SIP outbound proxy** is set (neither outbound_proxy_IP nor outbound_proxy_hostname is set), the following parameter setting is invalid:

- If E_SIP_OutboundProxyTransport is set to ENUM_TCP, **gc_Start( )** returns an IPERR_BAD_PARM error.

4.2 Using Fast Start and Slow Start Setup

Fast start and slow start are supported in both the H.323 and SIP protocols. Fast start connection is preferable to slow start connection because fewer network round trips are required to set up a call and the local exchange can generate messages when circumstances prevent a connection to the endpoint.

In H.323, fast start and slow start setup are supported depending on the version of H.323 standard supported at the remote side. If the remote side supports H.323 version 2 or above, fast start setup can be used; otherwise, a slow start setup is used. Fast start connection reduces the time required to set up a call to one round-trip delay following the H.225 TCP connection. The concept is to include all the necessary parameters for the logical channel to be opened (H.245 information) in the Setup message. The logical channel information represents a set of supported capabilities from which the remote end can choose the most appropriate capability. If the remote side decides to use fast start connection, it returns the desired logical channel parameters in the Alerting, Proceeding, or Connect messages.

Note: In an H.323 fast start call, the fast start element (FSE) is included in outgoing H.225 PROCEEDING / ALERTING only when the user explicitly specifies the coders. If no coder is specified (either a preferred coder or “don't care”) before **gc_CallAck( )** and **gc_AcceptCall( )** the FSE is not sent out until CONNECT (i.e., after **gc_AnswerCall( )**).

In SIP, fast start and slow start setup are also supported. In slow start setup, the INVITE message has no Session Description Protocol (SDP) and the remote side therefore proposes the session attributes in the SDP of the ACK message.

In Global Call, fast start and slow start connection are supported on a call-by-call basis. Fast start connection is used by default, but slow start connection can be forced by including the IPPARM_CONNECTIONMETHOD parameter ID with a value of IP_CONNECTIONMETHOD_SLOWSTART in the ext_datap field (of type GC_PARM_BLK) in the GCLIB_MAKECALL_BLK structure associated with a call. The following code segment shows how to specify a slow start connection explicitly by including the IPPARM_CONNECTIONMETHOD parameter ID when populating the ext_datap field:

```
gc_util_insert_parm_val(&libBblock.ext_datap,
                        IPSET_CALLINFO,
                        IPPARM_CONNECTIONMETHOD,
                        sizeof(char),
                        IP_CONNECTIONMETHOD_SLOWSTART);
```

In addition, the IPPARM_CONNECTIONMETHOD parameter ID can be set to a value of IP_CONNECTIONMETHOD_FASTSTART to force a fast start connection on a line device configured to use a slow start connection (using **gc_SetUserInfo( )** with a **duration** parameter of GC_ALLCALLS).

Note: In SIP, only the calling side can choose fast start or slow start, unlike H.323 where both sides can select either fast start or slow start.

4.3 Setting Call-Related Information

Global Call allows applications to set many items of call-related information. The following topics are presented in this section:

- Overview of Setting Call-Related Information
- Setting Coder Information
- Specifying Nonstandard Data Information When Using H.323
- Specifying Nonstandard Control Information When Using H.323
- Setting and Retrieving Disconnect Cause or Reason Values
- Setting Busy Reason Codes

4.3.1 Overview of Setting Call-Related Information

Table 1 summarizes the types of information elements that can be specified, the corresponding set IDs and parameter IDs used to set the information, the functions that can be used to set the information, and an indication of whether the information is supported when using H.323, SIP, or both. For more information on the various parameters, refer to the corresponding parameter set reference section in Chapter 8, "IP-Specific Parameters".

Table 1. Summary of Call-Related Information that can be Set

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	SIP/ H.323
Bearer Capability IE	IPSET_CALLINFO • IPPARM_BEARERCAP	**gc_SetUserInfo( )** (GC_SINGLECALL only)	H.323 only

Call ID (GUID)	IPSET_CALLINFO
	• IPPARM_CALLID
	Note: Setting the Call ID must be done judiciously because it might affect the call control implementation supported by the stack. The Call ID should be treated as a GUID and should be unique at all times.
		gc_SetUserInfo( ) (GC_SINGLECALL only)
		gc_MakeCall( )
			both

Coder Information †	GCSET_CHAN_CAPABILITY
	• IPPARM_LOCAL_CAPABILITY
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

Conference Goal	IPSET_CONFERENCE
	• IPPARM_CONFERENCE_GOAL
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323 only

Connection Method	IPSET_CALLINFO
	• IPPARM_CONNECTIONMETHOD
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

† If no transmit or receive coder type is specified, any supported coder type is accepted. The default is "don't care"; that is, any media coder supported by the platform is valid.
†† The duration parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
††† On the terminating side, can only be set using **gc_SetConfigData( )** on a board device. See Section 4.19, "Enabling and Disabling H.245 Tunneling", on page 206 for more information.

Table 1. Summary of Call-Related Information that can be Set (Continued)

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	SIP/ H.323
DTMF Support	IPSET_DTMF • IPPARM_SUPPORT_DTMF_BITMASK	**gc_SetConfigData( )** **gc_SetUserInfo( )** ††	both

Display Information	IPSET_CALLINFO
	• IPPARM_DISPLAY
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

Enabling/Disabling Unsolicited Events	IPSET_EXTENSIONEVT_MSK • GCACT_ADDMSK • GCACT_SETMSK • GCACT_SUBMSK	**gc_SetConfigData( )**	both
Facility IE	IPSET_CALLINFO • IPPARM_FACILITY	**gc_SetUserInfo( )** (GC_SINGLECALL only)	H.323 only

MediaWaitFor Connect	IPSET_CALLINFO
	• IPPARM_MEDIAWAITFORCONNECT
		gc_SetUserInfo( ) (GC_SINGLECALL only)
		gc_MakeCall( )
			H.323 only

Nonstandard Control Information	IPSET_NONSTANDARDCONTROL
	Either:
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_NONSTANDARDDATA_OBJID
	or
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_H221NONSTANDARD
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323 only

Nonstandard Data	IPSET_NONSTANDARDDATA
	Either:
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_NONSTANDARDDATA_OBJID
	or
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_H221NONSTANDARD
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323 only

Phone List	IPSET_CALLINFO
	• IPPARM_PHONELIST
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

Presentation Indicator	IPSET_CALLINFO
	• IPPARM_PRESENTATION_IND
		gc_SetUserInfo( ) (GC_SINGLECALL only)
		gc_MakeCall( )
			H.323 only

† If no transmit or receive coder type is specified, any supported coder type is accepted. The default is “don’t care”; that is, any media coder supported by the platform is valid.
†† The duration parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
††† On the terminating side, can only be set using **gc_SetConfigData( )** on a board device. See Section 4.19, “Enabling and Disabling H.245 Tunneling”, on page 206 for more information.

Table 1. Summary of Call-Related Information that can be Set (Continued)

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	SIP/ H.323
SIP Message Information Fields	IPSET_SIP_MSGINFO • IPPARM_CALLID_HDR • IPPARM_CONTACT_DISPLAY • IPPARM_CONTACT_URI • IPPARM_DIVERSION_URI • IPPARM_FROM_DISPLAY • IPPARM_REFERRED_BY • IPPARM_REPLACES • IPPARM_REQUEST_URI • IPPARM_TO_DISPLAY	**gc_SetUserInfo( )** (GC_SINGLECALL only)	SIP only
T.38 Fax device association or disassociation with Media device	IPSET_FOIP • IPPARM_T38_CONNECT • IPPARM_T38_DISCONNECT	**gc_SetUserInfo( )** †	both

Tunnelling†††	IPSET_CALLINFO
	• IPPARM_H245TUNNELING
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323 only

Type of Service (ToS)	IPSET_CONFIG • IPPARM_CONFIG_TOS	**gc_SetUserInfo( )** †† **gc_MakeCall( )**	H.323 only

User to User Information	IPSET_CALLINFO
	• IPPARM_USERUSER_INFO
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323 only

Vendor Information	IPSET_VENDORINFO • IPPARM_H221NONSTD • IPPARM_VENDOR_PRODUCT_ID • IPPARM_VENDOR_VERSION_ID	**gc_SetConfigData( )**	H.323 only

† If no transmit or receive coder type is specified, any supported coder type is accepted. The default is “don’t care”; that is, any media coder supported by the platform is valid.
†† The duration parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
††† On the terminating side, can only be set using **gc_SetConfigData( )** on a board device. See Section 4.19, “Enabling and Disabling H.245 Tunneling”, on page 206 for more information.

4.3.1.1 Setting Call Parameters on a System-Wide Basis

Use the **gc_SetConfigData( )** function to configure call-related parameters including coder information. The values set by the **gc_SetConfigData( )** function are used by the call control library as default values for each line device.

See Section 7.3.24, “gc_SetConfigData( ) Variances for IP”, on page 334 for more information about the values of function parameters to set in this context.

4.3.1.2 Setting Call Parameters on a Per Line Device Basis

The **gc_SetUserInfo( )** function (with the **duration** parameter set to GC_ALLCALLS) can be used to set the values of call-related parameters on a per line-device basis. The values set by **gc_SetUserInfo( )** become the new default values for the specified line device and are used by all subsequent calls on that device. See Section 7.3.25, "gc_SetUserInfo( ) Variances for IP", on page 337 for more information about the values of function parameters to set in this context.

4.3.1.3 Setting Call Parameters on a Per Call Basis

There are two ways to set call parameters on a per-call basis:

- Using **gc_SetUserInfo( )** with the **duration** parameter set to GC_SINGLECALL
- Using **gc_MakeCall( )**

Setting Per-Call Call Parameters Using gc_SetUserInfo( )

The **gc_SetUserInfo( )** function (with the **duration** parameter set to GC_SINGLECALL) can be used to set call parameter values for a single incoming call. At the end of the call, the values set as default values for the specified line device override these values. This is useful since the **gc_AnswerCall( )** function does not have a parameter to specify a GC_PARM_BLK.

If a **gc_MakeCall( )** function is issued after the **gc_SetUserInfo( )**, the values specified in the **gc_MakeCall( )** function override the values specified by the **gc_SetUserInfo( )** function. See Section 7.3.25, "gc_SetUserInfo( ) Variances for IP", on page 337 for more information about the values of function parameters to set in this context.

Setting Per-Call Call Parameters Using gc_MakeCall( )

The **gc_MakeCall( )** function can be used to set call parameter values for a call. The values set are only valid for the duration of the current call. At the end of the call, the values set as default values for the specified line device override the values specified by the **gc_MakeCall( )** function.

See Section 7.3.16, "gc_MakeCall( ) Variances for IP", on page 311 for more information about the values of function parameters to set in this context.

4.3.2 Setting Coder Information

Terminal capabilities are exchanged during call establishment. The terminal capabilities are sent to the remote side as notification of coder supported.

Coder information can be set in the following ways:

- On a system wide basis using **gc_SetConfigData( )**.
- On a per line device basis using **gc_SetUserInfo( )** with a **duration** parameter value of GC_ALLCALLS.
- On a per call basis using **gc_MakeCall( )** or **gc_SetUserInfo( )** with a **duration** parameter value of GC_SINGLECALL.

In each case, a GC_PARM_BLK is set up to contain the coder information. The GC_PARM_BLK must contain the GCSET_CHAN_CAPABILITY parameter set ID with the IPPARM_LOCAL_CAPABILITY parameter ID, which is of type IP_CAPABILITY.

Possible values for fields in the IP_CAPABILITY structure are:

capability

Specifies the coder type from among the types supported by the particular IP telephony platform; see Table 2 for platform-specific coder types. The following values are defined for the capability field:

- GCCAP_AUDIO_g711Alaw64k
- GCCAP_AUDIO_g711Ulaw64k
- GCCAP_AUDIO_g729AnnexA
- GCCAP_AUDIO_g729AnnexAwAnnexB
- GCCAP_AUDIO_NO_AUDIO
- GCCAP_DATA_t38UDPFax
- GCCAP_dontCare – The complete list of coders supported by a product is used when negotiating the coder type to be used. If multiple variations of the same coder are supported by a product, the underlying call control library offers the preferred variant only. For example, if G.711 10ms, 20ms, and 30ms are supported, only the preferred variant, G.711 20 ms, is included.

type

One of the following:

- GCCAPTYPE_AUDIO
- GCCAPTYPE_RDATA

direction

One of the following:

- IP_CAP_DIR_LCLTRANSMIT – transmit capability
- IP_CAP_DIR_LCLRECEIVE – receive capability
- IP_CAP_DIR_LCLRXTX – transmit and receive capability (T.38 only)

Note: It is recommended to specify both the transmit and receive capabilities.

payload_type

Not supported. The currently supported coders have static (pre-assigned) payload types defined by standards.

extra

Reference to a data structure of type IP_AUDIO_CAPABILITY, which contains the following two fields:

- frames_per_packet – The number of frames per packet.

Note: For G.711 coders, the extra.frames_per_packet field is the frame size (in ms).

- VAD – Enables or disables VAD. Values: GCPV_DISABLE, GCPV_ENABLE, GCCAP_dontCare

Note: Applications must explicitly set this field to GCPV_ENABLE for the coders that implicitly support only VAD, such as GCCAP_AUDIO_g729AnnexAwAnnexB.

See the reference page for IP_CAPABILITY on page 385 for more information.

Table 2 shows the coders that are supported when using the Global Call API with Intel NetStructure Host Media Processing (HMP) software.

Table 2. Coders Supported for Host Media Processing (HMP)

Coder and Rate	Global Call # Define	Frames Per Packet (fpp) and Frame Size (ms)	VAD Support
G.711 A-law	GCCAP_AUDIO_g711Alaw64k	Frame Size[1]: 10, 20, or 30 ms (Frames Per Packet: fixed at 1 fpp)	Not supported; must be explicitly disabled
G.711 mu-law	GCCAP_AUDIO_g711Ulaw64k	Frame Size[1]: 10, 20, or 30 ms (Frames Per Packet: fixed at 1 fpp)	Not supported; must be explicitly disabled
G.723.1 5.3 kbps	GCCAP_AUDIO_g7231_5_3k	Frames Per Packet: 2 or 3 (Frame Size: fixed at 30 ms)	Supported
G.723.1, 6.3 kbps	GCCAP_AUDIO_g7231_6_3k	Frames Per Packet: 2 or 3 (Frame Size: fixed at 30 ms)	Supported
G.729a	GCCAP_AUDIO_g729AnnexA	Frames Per Packet: 2, 3, or 4 (Frame Size: fixed at 10 ms)	Not supported; must be explicitly disabled
G.729a+b	GCCAP_AUDIO_g729AnnexAwAnnexB	Frames Per Packet: 2, 3, or 4 (Frame Size: fixed at 10 ms)	Must be enabled [2]
T.38	GCCAP_DATA_t38UDPFax	Not applicable	Not applicable

Note:
1. For G.711 coders, the frame size value (not the frames per packet value) is specified in the frames_per_pkt field of the IP_AUDIO_CAPABILITY structure. See the reference page for IP_AUDIO_CAPABILITY on page 383 for more information.
2. Applications must explicitly specify VAD support even though G.729a+b implicitly supports VAD.

4.3.2.1 Specifying Media Capabilities Before Connection

Applications can only specify media capabilities before initial call connection. For an outbound call, capabilities must be set before or with the **gc_MakeCall( )**. For inbound calls, capabilities must be set before or with the **gc_AnswerCall( )**, but it is recommended that they be set before **gc_AcceptCall( )**. Capability types can be GCCAPTYPE_AUDIO and/or GCCAPTYPE_RDATA. The result capabilities set by applications are listed in the Table 3.

Table 3. Capabilities Set by Application

GCCAPTYPE_AUDIO capability set by application	GCCAPTYPE_RDATA capability set by application	Result Capability
Not set	Not set	Support all audio capabilities during initial call connection. Support only switching between audio and T.38 during call connection. Do not support switching between different audio capabilities during call connection.
One or more GCCAP_AUDIO_XXX	Not Set	Support only audio capabilities specified no T.38 capability. Do not support switching between different audio capabilities during call connection.
Not Set	GCCAP_ DATA_t38UDPFax	Support only T.38 fax capability, no audio capability.

Table 3. Capabilities Set by Application (Continued)

GCCAPTYPE_AUDIO capability set by application	GCCAPTYPE_RDATA capability set by application	Result Capability
One or more GCCAP_AUDIO_XXX	GCCAP_ DATA_t38UDPFax	Support only audio capabilities specified during initial call connection. Support only switching between audio and T.38 during call connection. Do not support switching between different audio capabilities.
GCCAP_dontCare	Not Set	Support all audio capabilities during initial call connection. Do not support switching between different audio capabilities during call connection. Do not support T.38.
GCCAP_dontCare	GCCAP_ DATA_t38UDPFax	Support all audio capabilities during initial call connection. Support only switching between audio and T.38 during call connection. Do not support switching between different audio capabilities during call connection.

4.3.2.2 Resource Allocation When Using Low-Bit Rate Coders

The number of resources available when using G.723 and G.729 coders is limited. When all resources are consumed, depending on the requirements of the application, different behavior may be observed as follows:

- If the application specifies only G.723 and/or G.729 audio coders before **gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, or **gc_AnswerCall( )**, the result is a function failure with an error code of IPERR_TXRXRESOURCESINSUFF.
- If the application specifies G.711 with G.723 and/or G.729 audio coders, only the G.711 coder will be provided in the capability set sent to the remote endpoint.
- If the application does not explicitly specify any audio capability, then the G.711 (both A-law and u-law) coders are included in the capability set sent to the remote endpoint.

LBR coder resources are only released when **gc_ReleaseCallEx( )** is used, regardless of whether the resource was negotiated or not.

4.3.3 Specifying Nonstandard Data Information When Using H.323

To specify Nonstandard Data, set up the GC_PARM_BLK pointed by the **infoparmblkp** function parameter with the IPSET_NONSTANDARDDATA parameter set ID, and one of two possible combinations of parameter IDs. First set the parameter ID (maximum length MAX_NS_PARM_DATA_LENGTH or128). Then set either of the following two parameter IDs, depending on the type of object identifier to use:

- IPPARM_NONSTANDARDDATA_OBJID. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).
- IPPARM_H221NONSTANDARD

See Section 8.2.18, “IPSET_NONSTANDARDDATA”, on page 368 for more information.

The following code example shown how to set nonstandard data elements:

```
IP_H221NONSTANDARD appH221NonStd;
appH221NonStd.country_code = 181;
appH221NonStd.extension = 31;
appH221NonStd.manufacturer_code = 11;
char* pData = "Data String";
char* pOid = "1 22 333 4444";
choiceOfNSData = 1;/* App decides which type of object identifier to use */

/* setting NS Data */
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_NONSTANDARDDATA,
                        IPPARM_NONSTANDARDDATA_DATA,
                        (unsigned char)(strlen(pData)+1),
                        pData);

   if (choiceOfNSData) /* App decides the CHOICE of OBJECTIDENTIFIER.
                          It cannot set both objid & H221 */
   {
      gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_NONSTANDARDDATA,
                              IPPARM_H221NONSTANDARD,
                              (unsigned char)sizeof(IP_H221NONSTANDARD),
                              &appH221NonStd);
   }

   else
   {
      gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_NONSTANDARDDATA,
                              IPPARM_NONSTANDARDDATA_OBJID,
                              (unsigned char)(strlen(pOid)+1),
                              pOid);
   }
```

4.3.4 Specifying Nonstandard Control Information When Using H.323

To specify Nonstandard Control information, use the **gc_SetUserInfo( )** function with a **duration** parameter set to GC_SINGLECALL to set nonstandard control information. If the **duration** parameter is set to GC_ALLCALLS, the function will fail.

Set up the GC_PARM_BLK pointed by the **infoparmblkp** function parameter with the IPSET_NONSTANDARDCONTROL parameter set ID and one of two combinations of parameter IDs. First set the IPPARM_NONSTANDARDDATA_DATA parameter ID (maximum length is MAX_NS_PARM_DATA_LENGTH or 128). Then set either of the following parameter IDs according to which type of object identifier to use:

- IPPARM_NONSTANDARDDATA_OBJID. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).
- IPPARM_H221NONSTANDARD

See Section 8.2.17, “IPSET_NONSTANDARDCONTROL”, on page 368 for more information.

The following code example shows how to set nonstandard data elements:

```
IP_H221NONSTANDARD appH221NonStd;
appH221NonStd.country_code = 181;
appH221NonStd.extension = 31;
appH221NonStd.manufacturer_code = 11;
char* pControl = "Control String";
char* pOid = "1 22 333 4444";
choiceOfNSControl = 1; /* App decides which type of object identifier to use */

/* setting NS Control */
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_NONSTANDARDCONTROL,
                        IPPARM_NONSTANDARDDATA_DATA,
                        (unsingned char)(strlen(pControl)+1),
                        pControl);

   if (choiceOfNSControl)  /* App decide the CHOICE of OBJECTIDENTIFIER.
                              It cannot set both objid & h221 */
   {
      gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_H221NONSTANDARD,
                              (unsingned char)sizeof(IP_H221NONSTANDARD),
                              &appH221NonStd);
   }

   else
   {
      gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_NONSTANDARDDATA_OBJID,
                              (unsingned char)(strlen(pOid)+1),
                              pOid);
   }
```

4.3.5 Setting and Retrieving Disconnect Cause or Reason Values

Use the **cause** parameter in the **gc_DropCall( )** function to specify a disconnect reason/cause to be sent to the remote endpoint.

Note: When using SIP, reasons are only supported when a call is disconnected while in the Offered state.

Use the **gc_ResultInfo( )** function to get the reason/cause of a GCEV_DISCONNECTED event. This reason/cause could be sent from the remote endpoint or it could be the result of an internal error.

IP-specific reason/cause values are specified in the eIP_EC_TYPE enumerator defined in the *gcip_defs.h* header file.

4.3.6 Setting Busy Reason Codes

Both SIP and H.323 define request response codes that can be included in the failure response messages that are sent when a local system cannot take additional incoming sessions. Global Call allows applications to set SIP and H.323 busy code values on a virtual board level.

SIP and H.323 busy codes are configured independently, and the configuration of each can be changed at any time. The busy codes are configured by calling **gc_SetConfigData( )** using the following parameter set ID and parameter ID:

- for SIP: IPSET_SIP_RESPONSE_CODE and IPPARM_BUSY_REASON; see Section 8.2.24, "IPSET_SIP_RESPONSE_CODE", on page 374.
- for H.323: IPSET_H323_RESPONSE_CODE and IPPARM_BUSY_CAUSE; see Section 8.2.8, "IPSET_H323_RESPONSE_CODE", on page 363.

4.3.6.1 Setting SIP Busy Code

For SIP, RFC3261 defines three applicable busy codes:

480 Temporarily Unavailable

The callee's end system was contacted successfully, but the callee is currently unavailable. For example, the callee may be not logged in, may be in a state that precludes communication, or may have activated the "do not disturb" feature. This busy code is also returned by a redirect or proxy server that recognizes the user identified by the Request-URI but does not currently have a valid forwarding location for that user.

486 Busy Here

The callee's end system was contacted successfully, but the callee is currently not willing or able to take additional calls at this end system. This response should be used if the user could be available elsewhere.

600 Busy Everywhere

The callee's end system was contacted successfully, but the callee is busy and does not wish to take the call at this time. This response should be used if the callee knows that no other end system will be available to accept this call.

By default, Global Call automatically responds with a 486 Busy Here when additional incoming call requests arrive after the maximum number of SIP calls per virtual board has been reached. A 480 Temporarily Unavailable or 600 Busy Everywhere reason code can be used instead of the 486 Busy Here if the application explicitly configures the busy code.

To configure the SIP busy reason code, call **gc_SetConfigData( )** using the parameter set ID IPSET_SIP_RESPONSE_CODE and the parameter ID IPPARM_BUSY_REASON as shown in the following code snippet:

```
#include "gclib.h"
   .
   .
/* configure SIP Busy Reason Code to 480 Temporarily Available */

GC_PARM_BLKP pParmBlock = NULL;

gc_util_set_insert_parm_val(&pParmBlock,
                            IPSET_SIP_RESPONSE_CODE,
                            IPPARM_BUSY_REASON,
                            sizeof(unsigned short),
                            IPEC_SIPReasonStatus480TemporarilyUnavailable);

gc_SetConfigData(GCTGT_CCLIB_NETIF, board, pParmBlock,
                 0, GCUPDATE_IMMEDIATE, &t, EV_ASYNC);

gc_util_delete_parm_blk(pParmBlock);
```

4.3.6.2 Setting H.323 Busy Code

ITU Recommendation Q.850 defines cause codes that are used for H.323. Among the applicable busy cause definitions are:

Cause 34: No circuit/channel available
Indicates there is no appropriate circuit/channel currently available to handle the call.

Cause 47: Resource unavailable/unspecified
Indicates the resource is unavailable when no other cause values in the resource class applies.

To configure the H.323 busy reason code, call **gc_SetConfigData( )** using the parameter set ID IPSET_H323_RESPONSE_CODE and the parameter ID IPPARM_BUSY_CAUSE as shown in the following code snippet:

```
#include "gclib.h"
   .
   .
/* configure H.323 Busy Reason Code to 34 -  "No Circuit/Channel Available" */

GC_PARM_BLKP pParmBlock = NULL;

gc_util_set_insert_parm_val(&pParmBlock,
                            IPSET_H323_RESPONSE_CODE,
                            IPPARM_BUSY_CAUSE,
                            sizeof(unsigned short),
                            IPEC_Q931Cause34NoCircuitChannelAvailable);

gc_SetConfigData(GCTGT_CCLIB_NETIF, board, pParmBlock,
                 0, GCUPDATE_IMMEDIATE, &t, EV_ASYNC);

gc_util_delete_parm_blk(pParmBlock);
```

4.4 Retrieving Current Call-Related Information

To support large numbers of channels, the call control library must perform all operations in asynchronous mode. To support this, an extension function variant allows the retrieval of a parameter as an asynchronous operation.

The retrieval of call-related information is a four step process:

1. Set up a GC_PARM_BLK that identifies which information is to be retrieved. The GC_PARM_BLK includes GC_PARM_DATA blocks. The GC_PARM_DATA blocks specify only the Set_ID and Parm_ID fields, that is, the value_size field is set to 0. The list of GC_PARM_DATA blocks indicate to the call control library the parameters to be retrieved.
2. Use the **gc_Extension( )** function to request the data. The parameters for this call should be specified as follows:
   - **target_type** should be GCTGT_GCLIB_CRN
   - **target_id** should be the actual CRN
   - **ext_id** (extension ID) should be set to IPEXTID_GETINFO
   - **parmblkp** should point to the GC_PARM_BLK set up in step 1
   - **mode** should be set to EV_ASYNC (asynchronous)

3. A GCEV_EXTENSIONCMPLT event is generated in response to the **gc_Extension( )** request. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSIONCMPLT event is a pointer to an EXTENSIONEVTBLK structure that contains a GC_PARM_BLK with the requested call-related information.
4. Extract the information from the GC_PARM_BLK associated with the GCEV_EXTENSIONCMPLT event. In this case, the GC_PARM_BLK contains real data; that is, the value_size field is not 0, and includes the size of the data following for each parameter requested.

Note: When an application on H.323 is using **gc_Extension( )** to extract information from a GCEV_OFFERED event, the application must ensure that it acknowledges the call within 8 seconds to prevent the offering side from timing out. The timer can be extended by sending PROCEEDING (by calling **gc_CallAck( )**) or ALERTING (by calling **gc_AcceptCall( )**) before extracting the information.

Table 4 shows the parameters that can be retrieved and when the information should be retrieved. The table also identifies which information can be retrieved when using H.323 and which information can be retrieved using SIP. For more information on individual parameters, refer to the corresponding parameter set reference section in Chapter 8, “IP-Specific Parameters”.

Table 4. Retrievable Call Information

Parameter	Set ID and Parameter ID(s)	When Information Can Be Retrieved	Datatype in value_buf Field (see Note 1)	SIP/ H.323
Call ID	IPSET_CALLINFO • IPPARM_CALLID	Any state after Offered or Proceeding	For SIP: string, max. length = MAX_IP_SIP_CALLID_LENGTH For H.323: array of octets, length = MAX_IP_H323_CALLID_LENGTH If protocol is unknown, MAX_IP_CALLID_LENGTH defines the maximum Call ID length for any possible protocol.	both
Bearer Capability IE	IPSET_CALLINFO • IPPARM_BEARERCAP	After Offered	String, max. length = 255	H.323 only
Call Duration	IPSET_CALLINFO • IPPARM_CALL_DURATION	After Disconnected, before Idle	Unsigned long (value in ms)	H.323 only

Notes:
1. This field is the value_buf field in the GC_PARM_DATA structure associated with the GCEV_EXTENSIONCMPLT event generated in response to the **gc_Extension( )** function requesting the information.
2. Display information, user to user information, phone list, nonstandard data, vendor information and nonstandard control information, and H221 nonstandard information may not be present.
3.Vendor information is included in a Q931 SETUP message received from a peer.
4. The nonstandard object id and nonstandard data parameters described here refer to nonstandard data contained in a SETUP message for example. This should not be confused with the nonstandard data included in protocol messages sent using **gc_Extension( )** which can be retrieved from the metaevent associated with a GCEV_EXTENSION event.

Table 4. Retrievable Call Information (Continued)

Parameter	Set ID and Parameter ID(s)	When Information Can Be Retrieved	Datatype in value_buf Field (see Note 1)	SIP/ H.323
Conference Goal	IPSET_CONFERENCE • IPPARM_CONFERENCE_GOAL	Any state after Offered or Proceeding	Uint[8]	H.323 only
Conference ID	IPSET_CONFERENCE • IPPARM_CONFERENCE_ID	Any state after Offered or Proceeding	char*, max. length = IP_CONFERENCE_ID_ LENGTH (16)	H.323 only
Display Information	IPSET_CALLINFO • IPPARM_DISPLAY	Any state after Offered or Proceeding	char*, max. length = MAX_DISPLAY_ LENGTH (82), null-terminated	both
Facility IE	IPSET_CALLINFO • IPPARM_FACILITY	After Offered (SETUP message), Connected (CONNECT message), or the reception of a Facility message	String, max. length = 255	H.323 only
Nonstandard Control	IPSET_NONSTANDARDCONTROL • IPPARM_ NONSTANDARDDATA_DATA • IPPARM_ NONSTANDARDDATA_OBJID or • IPPARM_H221NONSTANDARD	See Section 4.4.1, “Retrieving Nonstandard Data From Protocol Messages (H.323)”, on page 118 for more information.	char*, max. length = MAX_NS_PARM_ DATA_LENGTH (128) char*, max. length = MAX_NS_PARM_ OBJID_LENGTH (40)	H.323 only
Nonstandard Data	IPSET_NONSTANDARDDATA • IPPARM_ NONSTANDARDDATA_DATA • IPPARM_ NONSTANDARDDATA_OBJID or • IPPARM_H221NONSTANDARD	See Section 4.4.1, “Retrieving Nonstandard Data From Protocol Messages (H.323)”, on page 118 for more information.	char*, max. length = MAX_NS_PARM_ DATA_LENGTH (128) char*, max. length = MAX_NS_PARM_ OBJID_LENGTH (40)	H.323 only
Phone List	IPSET_CALLINFO • IPPARM_PHONELIST	Any state after Offered or Proceeding	char*, max. length = 131	both
User to User Information	IPSET_CALLINFO • IPPARM_USERUSER_INFO	Any state after Offered or Proceeding	char*, max. length = MAX_USERUSER_INFO_LENGTH (131 octets)	H.323 only

Notes:
1. This field is the value_buf field in the GC_PARM_DATA structure associated with the GCEV_EXTENSIONCMPLT event generated in response to the **gc_Extension( )** function requesting the information.
2. Display information, user to user information, phone list, nonstandard data, vendor information and nonstandard control information, and H221 nonstandard information may not be present.
3.Vendor information is included in a Q931 SETUP message received from a peer.
4. The nonstandard object id and nonstandard data parameters described here refer to nonstandard data contained in a SETUP message for example. This should not be confused with the nonstandard data included in protocol messages sent using **gc_Extension( )** which can be retrieved from the metaevent associated with a GCEV_EXTENSION event.

Table 4. Retrievable Call Information (Continued)

Parameter	Set ID and Parameter ID(s)	When Information Can Be Retrieved	Datatype in value_buf Field (see Note 1)	SIP/ H.323
Vendor Product ID	IPSET_VENDORINFO • IPPARM_ VENDOR_PRODUCT_ID	Any state after Offered or Proceeding	char*, max. length = MAX_PRODUCT_ ID_LENGTH (32)	H.323 only
Vendor Version ID	IPSET_VENDORINFO • IPPARM_ VENDOR_VERSION_ID	Any state after Offered or Proceeding	char*, max. length = MAX_VERSION_ ID_LENGTH (32)	H.323 only
H.221 Nonstandard Information	IPSET_VENDORINFO • IPPARM_H221NONSTD	Any state after Offered or Proceeding	IP_H221_ NONSTANDARD (see note 4)	H.323 only

Notes:
1. This field is the value_buf field in the GC_PARM_DATA structure associated with the GCEV_EXTENSIONCMPLT event generated in response to the **gc_Extension( )** function requesting the information.
2. Display information, user to user information, phone list, nonstandard data, vendor information and nonstandard control information, and H221 nonstandard information may not be present.
3.Vendor information is included in a Q931 SETUP message received from a peer.
4. The nonstandard object id and nonstandard data parameters described here refer to nonstandard data contained in a SETUP message for example. This should not be confused with the nonstandard data included in protocol messages sent using **gc_Extension( )** which can be retrieved from the metaevent associated with a GCEV_EXTENSION event.

If an attempt is made to retrieve information in a state in which the information is not available, no error is generated. The GC_PARM_BLK associated with the GCEV_EXTENSIONCMPLT event will not contain the requested information. If phone list and display information are requested and only phone list is available, then only phone list information is available in the GC_PARM_BLK. An error is generated if there is an internal error (such as memory cannot be allocated).

All call information is available until a **gc_ReleaseCallEx**( ) is issued.

4.4.1 Retrieving Nonstandard Data From Protocol Messages (H.323)

Any Q.931 message can include nonstandard data. The application can use the **gc_Extension**( ) function with and **ext_id** of IPEXTID_GETINFO to retrieve the data while a call is in any state. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. The information is included with the corresponding GCEV_EXTENSIONCMPLT termination event.

Note: When retrieving nonstandard data, it is only necessary to specify IPPARM_NONSTANDARDDATA_DATA in the extension request. It is not necessary to specify IPPARM_NONSTANDARDDATA_OBJID or IPPARM_H221NONSTANDARD. The call control library ensures that the GCEV_EXTENSIONCMPLT event includes the correct information.

4.4.2 Examples of Retrieving Call-Related Information

The following code demonstrates how to do the following:

- create a structure that identifies which information should be retrieved, then use the **gc_Extension( )** with an **extID** of IPEXTID_GETINFO to issue the request
- extract the data from a structure associated with the GCEV_EXTENSIONCMPLT event received as a termination event to the **gc_Extension( )** function

Similar code can be used when using SIP, except that the code must include only information parameters supported by SIP (see Table 4, "Retrievable Call Information", on page 116).

Specifying Call-Related Information to Retrieve

The following function shows how an application can construct and send a request to retrieve call-related information.

```
int getInfoAsync(CRN crn)
{
   GC_PARM_BLKP gcParmBlk = NULL;
   GC_PARM_BLKP retParmBlk;
   int frc;

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_PHONELIST,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_CALLID,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CONFERENCE,
                                 IPPARM_CONFERENCE_ID,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CONFERENCE,
                                 IPPARM_CONFERENCE_GOAL,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }
```

```
frc = gc_util_insert_parm_val(&gcParmBlk,
                              IPSET_CALLINFO,
                              IPPARM_DISPLAY,
                              sizeof(int),1);
if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}

frc = gc_util_insert_parm_val(&gcParmBlk,
                              IPSET_CALLINFO,
                              IPPARM_USERUSER_INFO,
                              sizeof(int),1);

if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}

frc = gc_util_insert_parm_val(&gcParmBlk,
                              IPSET_VENDORINFO,
                              IPPARM_VENDOR_PRODUCT_ID,
                              sizeof(int),1);
if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}

frc = gc_util_insert_parm_val(&gcParmBlk,
                              IPSET_VENDORINFO,
                              IPPARM_VENDOR_VERSION_ID,
                              sizeof(int),1);
if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}

frc = gc_util_insert_parm_val(&gcParmBlk,
                              IPSET_VENDORINFO,
                              IPPARM_H221NONSTD,
                              sizeof(int),1);
if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}

frc = gc_util_insert_parm_val(&gcParmBlk,/* NS Data: setting this IPPARM implies
                                             retrieval of the complete element */
                              IPSET_NONSTANDARDDATA,
                              IPPARM_NONSTANDARDDATA_DATA,
                              sizeof(int),1);
if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}

frc = gc_util_insert_parm_val(&gcParmBlk,/* NS Control: setting this IPPARM implies
                                             retrieval of the complete element */
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_NONSTANDARDDATA_DATA,
                              sizeof(int),1);
if (GC_SUCCESS != frc)
{
   return GC_ERROR;
}
```

```
   frc = gc_Extension(GCTGT_GCLIB_CRN,
                      crn,
                      IPEXTID_GETINFO,
                      gcParmBlk,
                      &retParmBlk,
                      EV_ASYNC);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   gc_util_delete_parm_blk(gcParmBlk);
   return GC_SUCCESS;
}
```

Extracting Call-Related Information Associated with an Extension Event

The following code demonstrates how an application can extract call information when a GCEV_EXTENSIONCMPLT event is received as a result of a request for call-related information.

```
int OnExtensionAndComplete(GC_PARM_BLKP parm_blk,CRN crn)
{
   GC_PARM_DATA  *parmp = NULL;
   parmp = gc_util_next_parm(parm_blk,parmp);
   if (!parmp)
   {
      return GC_ERROR;
   }

   while (NULL != parmp)
   {
      switch (parmp->set_ID)
      {
         case IPSET_CALLINFO:
            switch (parmp->parm_ID)
            {
                case IPPARM_DISPLAY:
                   if(parmp->value_size != 0)
                   {
                      printf("\tReceived extension data DISPLAY: %s\n", parmp->value_buf);
                   }
                   break;

                case IPPARM_CALLID:
                   /* print the Call ID in parmp->value_buf as array of bytes */
                   for (int count = 0; count < parmp->value_size; count++)
                   {
                      printf("0x%2X ", value_buf[count]);
                   }
                   break;

                case IPPARM_USERUSER_INFO:
                   if(parmp->value_size != 0)
                   {
                      printf("\tReceived extension data UUI: %s\n", parmp->value_buf);
                   }
                   break;
```

```
       case IPPARM_PHONELIST:
          if(parmp->value_size != 0)
          {
             printf("\tReceived extension data PHONELIST: %s\n",
                      parmp->value_buf);
          }
          break;

       default:
          printf("\tReceived unknown CALLINFO extension parmID %d\n",
                   parmp->parm_ID);
          break;
   }/* end switch (parmp->parm_ID) for IPSET_CALLINFO */
   break;

case IPSET_CONFERENCE:
   switch (parmp->parm_ID)
   {
      case IPPARM_CONFERENCE_GOAL:
         if(parmp->value_size != 0)
         {
            printf("\tReceived extension data IPPARM_CONFERENCE_GOAL: %d\n",
                    (unsigned int)(*(parmp->value_buf)));
         }
         break;

      case IPPARM_CONFERENCE_ID:
         if(parmp->value_size != 0)
         {
            printf("\tReceived extension data IPPARM_CONFERENCE_ID: %s\n",
                    parmp->value_buf);
         }
         break;

      default:
         printf("\tReceived unknown CONFERENCE extension parmID %d\n",
                 parmp->parm_ID);
         break;
   }
   break;

case IPSET_VENDORINFO:
   switch (parmp->parm_ID)
   {
      case IPPARM_VENDOR_PRODUCT_ID:
         if(parmp->value_size != 0)
         {
            printf("\tReceived extension data  PRODUCT_ID %s\n", parmp->value_buf);
         }
         break;

      case IPPARM_VENDOR_VERSION_ID:
         if(parmp->value_size != 0)
         {
            printf("\tReceived extension data  VERSION_ID %s\n", parmp->value_buf);
         }
         break;

      case IPPARM_H221NONSTD:
      {
         if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
         {
            IP_H221NONSTANDARD *pH221NonStandard;
            pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
            printf("\tReceived extension data VENDOR H221NONSTD:
                    CC=%d, Ext=%d, MC=%d\n",
                    pH221NonStandard->country_code,
                    pH221NonStandard->extension,
                    pH221NonStandard->manufacturer_code);
          }
      }
          break;

      default:
         printf("\tReceived unknown VENDORINFO extension parmID %d\n",
                  parmp->parm_ID);
         break;
   }/* end switch (parmp->parm_ID) for IPSET_VENDORINFO */
   break;

case IPSET_NONSTANDARDDATA:
   switch (parmp->parm_ID)
   {
      case IPPARM_NONSTANDARDDATA_DATA:
         printf("\tReceived extension data (NSDATA) DATA: %s\n", parmp->value_buf);
         break;

      case IPPARM_NONSTANDARDDATA_OBJID:
         printf("\tReceived extension data (NSDATA) OBJID: %s\n", parmp->value_buf);
         break;

      case IPPARM_H221NONSTANDARD:
      {
         if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
         {
            IP_H221NONSTANDARD *pH221NonStandard;
            pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
            printf("\tReceived extension data (NSDATA) h221:CC=%d, Ext=%d, MC=%d\n",
                    pH221NonStandard->country_code,
                    pH221NonStandard->extension,
                    pH221NonStandard->manufacturer_code);
         }
      }
      break;

      default:
         printf("\tReceived unknown (NSDATA) extension parmID %d\n",
                 parmp->parm_ID);
         break;
   }
   break;

case IPSET_NONSTANDARDCONTROL:
   switch (parmp->parm_ID)
   {
      case IPPARM_NONSTANDARDDATA_DATA:
         printf("\tReceived extension data (NSCONTROL) DATA: %s\n",
                 parmp->value_buf);
         break;

      case IPPARM_NONSTANDARDDATA_OBJID:
         printf("\tReceived extension data (NSCONTROL) OBJID: %s\n",
                 parmp->value_buf);
         break;

      case IPPARM_H221NONSTANDARD:
      {
         if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
         {
            IP_H221NONSTANDARD *pH221NonStandard;
            pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
            printf("\tReceived extension data (NSCONTROL) h221:CC=%d, Ext=%d, MC=%d\n",
                    pH221NonStandard->country_code,
                             pH221NonStandard->extension,
                             pH221NonStandard->manufacturer_code);
                  }
               }
               break;

               default:
                  printf("\tReceived unknown (NSCONTROL) extension parmID %d\n",
                          parmp->parm_ID);
                  break;
            }
            break;

         case IPSET_MSG_Q931:
            switch (parmp->parm_ID)
            {
               case IPPARM_MSGTYPE:
                  switch ((*(int *)(parmp->value_buf)))
                  {
                     case IP_MSGTYPE_Q931_FACILITY:
                        printf("\tReceived extension data IP_MSGTYPE_Q931_FACILITY\n");
                        break;

                     default:
                        printf("\tReceived unknown MSG_Q931 extension parmID %d\n",
                                parmp->parm_ID);
                        break;
                  } /* end  switch ((int)(parmp->value_buf)) */
                    break;
            }/* end switch (parmp->parm_ID) for IPSET_MSG_Q931 */
            break;

         case IPSET_MSG_H245:
            switch (parmp->parm_ID)
            {
               case IPPARM_MSGTYPE:
                  switch ((*(int *)(parmp->value_buf)))
                  {
                     case IP_MSGTYPE_H245_INDICATION:
                        printf("\tReceived extension data IP_MSGTYPE_H245_INDICATION\n");
                        break;

                     default:
                        printf("\tReceived unknown MSG_H245 extension parmID %d\n",
                                parmp->parm_ID);
                        break;
                  }/* end  switch ((int)(parmp->value_buf)) */
                   break;
            }/* end switch (parmp->parm_ID) for IPSET_MSG_H245 */
            break;

         default:
            printf("\t Received unknown extension setID %d\n",parmp->set_ID);
            break;
      }/* end switch (parmp->set_ID) */

      parmp = gc_util_next_parm(parm_blk,parmp);
   }

   return GC_SUCCESS;
}
```

Note: IPPARM_CALLID is a set of bytes and should *not* be interpreted as a string.

Retrieving Call ID

The following code example illustrates how to request Call ID information via a **gc_Extension( )** call.

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_iptB1T1:P_SIP, :N_iptB1T2:P_SIP, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been called
 *    to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>

/*
 * Assume the 'crn' parameter holds the CRN associated
 * with the detected GCEV_OFFERED event.
 */
int request_call_info(CRN crn)
{
   int retval = GC_SUCCESS;
   GC_PARM_BLKP parmblkp = NULL;  /* input parameter block pointer */
   GC_PARM_BLKP retblkp = NULL;   /* pointer for output parameter block (unused) */
   GC_INFO gc_error_info;         /* GlobalCall error information data */

   /* allocate GC_PARM_BLK for Call-ID message parameter */
   gc_util_insert_parm_val(&parmblkp, IPSET_CALLINFO, IPPARM_CALLID, sizeof(int), 1);
   if (parmblkp == NULL)
   {
      /* memory allocation error */
      return(-1);
   }

   /* retrieve the Call-ID from the network */
   if (gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_GETINFO, parmblkp, &retblkp,
                    EV_ASYNC) != GC_SUCCESS)
   {
      /* process error return as shown */
      gc_ErrorInfo( &gc_error_info );
      printf ("Error: gc_Extension() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
              CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
              crn, gc_error_info.gcValue, gc_error_info.gcMsg, gc_error_info.ccLibId,
              gc_error_info.ccLibName, gc_error_info.ccValue, gc_error_info.ccMsg);
   }

   /* free the parameter block */
   gc_util_delete_parm_blk(parmblkp);

   return (retval);
}
```

Parsing Call ID Information (SIP Protocol)

The following code example illustrates how to parse the Call ID information retrieved via a **gc_Extension( )** call when the SIP protocol is being used.

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_iptB1T1:P_SIP, :N_iptB1T2:P_SIP, etc...)
 * 2. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been called
 *    to convert the event into metaevent.
 * 3. a GCEV_EXTENSIONCMPLT has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>

/* Assume the 'crn' parameter holds the CRN associated with the detected GCEV_EXTENSIONCMPLT
 * event, and the 'pEvt' parameter holds a pointer to the detected metaevent.
 */

int print_call_info(CRN crn, METAEVENT *pEvt)
{
   EXTENSIONEVTBLK *ext_data = NULL;
   GC_PARM_DATA *parmp = NULL;
   GC_PARM_BLK  *parm_blkp;

   if (pEvt)
   {
      if (pEvt->evttype == GCEV_EXTENSIONCMPLT)
      {
         ext_data = (EXTENSIONEVTBLK *)(pEvt->extevtdatap);
      }
   }

   if (!ext_data)
   {
      printf("\tNot a GCEV_EXTENSIONCMPLT event.\n");
      return GC_ERROR;
   }

   parm_blk = &(ext_data->parmblk);

   parmp = gc_util_next_parm(parm_blkp,parmp);
   if (!parmp)
   {
      printf("\tNo data returned in extension event for crn: 0x%lx\n", crn);
      return GC_ERROR;
   }

   while (NULL != parmp)
   {
      switch (parmp->set_ID)
      {
         case IPSET_CALLINFO:
            switch (parmp->parm_ID)
            {
               case IPPARM_CALLID:
                  if(parmp->value_size != 0)
                  {
                     /* Here's where we print the SIP Call ID */
                     printf("\tReceived extension data IPPARM_CALLID: %s\n",
                            parmp->value_buf);
                  }
                  break;
               default:
                  printf("\tReceived unexpected IPSET_CALLINFO parmID %d\n",
                               parmp->parm_ID);
                  break;
            } /* end switch (parmp->parm_ID) */
            break;

         default:
            printf("\t Received unexpected extension setID %d\n",
                   parmp->set_ID);
            break;

      } /* end switch (parmp->set_ID) */

      parmp = gc_util_next_parm(parm_blkp,parmp);
   } /* end while (parmp != NULL) */

   return GC_SUCCESS;
}
```

4.5 Setting and Retrieving Q.931 Message IEs

Global Call supports the setting and retrieving of Information Elements (IEs) in selected Q.931 messages. The level of support is described in the following topics:

- Enabling Access to Q.931 Message IEs
- Supported Q.931 Message IEs
- Setting Q.931 Message IEs
- Retrieving Q.931 Message IEs
- Common Usage Scenarios Involving Q.931 Message IEs

4.5.1 Enabling Access to Q.931 Message IEs

The ability to set and retrieve Q.931 message IEs is an optional feature that can be enabled or disabled at the time the **gc_Start( )** function is called.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions, which must be called before **gc_Start( )**, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The default value of the h323_msginfo_mask field in the IP_VIRTBOARD structure disables access to Q.931 message information elements. The default value of the h323_msginfo_mask field must therefore be overridden with the value IP_H323_MSGINFO_ENABLE for each IPT board device on which the feature is to be enabled. The following code snippet provides an example for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE; /* override Q.931 message default */
ip_virtboard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE; /* override Q.931 message default */
```

Note: Setting the h323_msginfo_mask field to a value of IP_H323_MSGINFO_ENABLE enables the setting or retrieving of all supported Q.931 message information elements collectively. Enabling and disabling access to individual Q.931 message information elements is **not** supported.

4.5.2 Supported Q.931 Message IEs

Table 5 shows the supported Q.931 message Information Elements (IEs), the parameter set ID and parameter ID that should be included in a GC_PARM_BLK when setting or retrieving the IEs, and the maximum allowed length of the IE value.

Table 5. Supported Q.931 Message Information Elements

IE Name	Set/Get	Set ID	Parameter ID	Maximum Length
Bearer Capability	Get and Set	IPSET_CALLINFO	IPPARM_BEARERCAP	255
Facility	Get and Set	IPSET_CALLINFO	IPPARM_FACILITY	255
Note: These parameters are character arrays with the maximum size of the array equal to the maximum length shown.				

4.5.3 Setting Q.931 Message IEs

The Global Call library supports the setting of the following Information Elements (IEs) in the following *outgoing* Q.931 messages:

- Bearer Capability IE in a SETUP message
- Facility IE in SETUP, CONNECT, and FACILITY messages

The **gc_SetUserInfo( )** function is used to set these IEs. The appropriate function parameters in this context are:

- **target_type** – GCTGT_GCLIB_CHAN
- **target_id** – line device
- **infoparmblkp** – a GC_PARM_BLK containing the IPSET_CALLINFO parameter set ID and one of the following parameter IDs:
  - – IPPARM_BEARERCAP
  - – IPPARM_FACILITY
- **duration** – GC_SINGLECALL (GC_ALLCALLS is not supported in this context)

4.5.4 Retrieving Q.931 Message IEs

The Global Call library supports the retrieval of the following Information Elements (IEs) from the following *incoming* Q.931 messages:

- Bearer Capability IE in a SETUP message
- Facility IE in SETUP, CONNECT, and FACILITY messages

Table 6 shows the Global Call events generated for incoming Q.931 messages and the parameter set ID and parameter IDs contained in the GC_PARM_BLK associated with each event.

Table 6. Supported IEs in Incoming Q.931 Messages

Incoming Q.931 Message	Global Call Event	Set ID	Parm ID
SETUP	GCEV_OFFERED	IPSET_CALLINFO	IPPARM_BEARERCAP
SETUP	GCEV_OFFERED	IPSET_CALLINFO	IPPARM_FACILITY
CONNECT	GCEV_CONNECTED	IPSET_CALLINFO	IPPARM_FACILITY
FACILITY	GCEV_EXTENSION with an ext_id of EXTID_RECEIVEMSG	IPSET_CALLINFO	IPPARM_FACILITY

Note: The application must retrieve the necessary IEs by copying them into its own buffer before the next call to **gc_GetMetaEvent**( ). Once the next **gc_GetMetaEvent**( ) call is issued, the Q.931 information is no longer available.

4.5.5 Common Usage Scenarios Involving Q.931 Message IEs

Table 7 shows how Global Call handles common scenarios that involve the use of Q.931 message IEs.

Table 7. Common Usage Scenarios Involving Q.931 Message IEs

Scenario	Behavior
The application invokes **gc_SetUserInfo( )** to set the Bearer Capability IE, then invokes **gc_MakeCall( )**	The Bearer Capability IE is parsed and added to the new outgoing SETUP message.
The application invokes **gc_SetUserInfo( )** to set the Facility IE, then invokes **gc_MakeCall( )**	The Facility IE is parsed and added to the new outgoing SETUP message.
The application invokes **gc_SetUserInfo( )** to set the Bearer Capability IE and the Facility IE, then invokes **gc_MakeCall( )**	The Bearer Capability IE and the Facility IE are parsed and added to the new outgoing SETUP message.
The application invokes **gc_SetUserInfo( )** to set the Facility IE, then invokes **gc_AnswerCall( )**	The Facility IE is parsed and added to the new outgoing CONNECT message.
The application invokes **gc_SetUserInfo( )** to set the Facility IE, then invokes **gc_Extension( )**	The Facility IE is parsed and added to the new outgoing FACILITY message.
The application receives a GCEV_OFFERED event with a Bearer Capability IE	The application retrieves the Bearer Capability IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_OFFERED event with a Facility IE	The application retrieves the Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_OFFERED event with Bearer Capability IE and Facility IE	The application retrieves the Bearer Capability IE and Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_CONNECTED event with a Facility IE	The application retrieves the Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_EXTENSION event with a Facility IE	The application retrieves the Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.

4.6 Setting and Retrieving SIP Message Header Fields

Global Call supports the setting and retrieving of SIP message header fields in various SIP message types, including INFO, INVITE, NOTIFY, OPTIONS, REFER, and SUBSCRIBE requests. These messages may be implicitly created and sent as a result of a Global Call function call (for example, **gc_MakeCall( )** sends INVITE, **gc_InvokeXfer( )** sends REFER, and **gc_ReqService( )** sends REGISTER), or they may be messages that are explicitly constructed and then sent via **gc_Extension( )**, such as INFO or NOTIFY requests. On the receiving side, the messages are passed to the application as GCEV_OFFERED, GCEV_REQ_XFER, GCEV_CALLINFO, or GEEV_EXTENSION events, depending on the SIP request type, with the message information contained in the metaevent. The SIP header access feature is described in the following topics:

- SIP Header Access Overview
- Enabling Access to SIP Header Information
- Enabling Long Header Values
- Registering SIP Header Fields to be Retrieved
- Setting SIP Header Fields for Outbound Messages
- Retrieving SIP Message Header Fields

4.6.1 SIP Header Access Overview

The Global Call library provides a uniform mechanism for setting SIP header fields in SIP messages using a single Global Call parameter definition (namely IPSET_SIP_MSGINFO / IPPARM_SIP_HDR). This new mechanism is intended to replace the previous header access mechanism that relied on header-specific parameter definitions. Among the advantages of the new mechanism are:

- supports all SIP header fields, including optional and proprietary fields
- directly extensible to support new header fields
- field content length can exceed 255 bytes
- uniform programming approach
- application can register to receive only the header fields it needs to access from incoming messages

Header Fields in Outgoing SIP Messages

After access to SIP message information has been enabled (see Section 4.6.2, “Enabling Access to SIP Header Information”, on page 137), an application sets SIP message header fields for outgoing messages by inserting the set ID / parm ID pair and the parameter value (header contents) for each field into a GC_PARM_BLK using **gc_util_insert_parm_ref_ex( )** or **gc_util_insert_parm_val( )**. The application uses the IPSET_SIP_MSGINFO parameter set ID and IPPARM_SIP_HDR parameter ID to set any SIP header field. The parameter value must start with the header name and must conform to the SIP specifications for content, syntax, and punctuation.

Once the GC_PARM_BLK is composed, the application can pass that parm block as a parameter in a Global Call function that directly sends a message (such as **gc_Extension( )**, which is used to send messages like INFO or OPTIONS, or **gc_ReqService( )**, which is used to send REGISTER requests) or can preset the header fields for the next message to be sent by calling the **gc_SetUserInfo( )** function. The use of **gc_SetUserInfo( )** to preset SIP message header fields for the next message is only recommended when using **gc_MakeCall( )**. For messages that are sent directly (using **gc_Extension( )**, for example) the preferred method is to pass the parameter block directly to the function, because a preset header is always used for the very next message sent, which might not be the intended message. When using **gc_SetUserInfo( )** to preset SIP message header fields, the **duration** parameter must be set to GC_SINGLECALL, and the information is not transmitted until the next Global Call function that sends a SIP message is issued.

Table 8 shows the relationship between some of the most common SIP header fields, the SIP messages that commonly use them, and the Global Call functions that are used to set the headers and send the message.

Note: The Global Call library handles the SIP Request-URI exactly like a standard SIP header field even though it is technically distinct from the header fields in a SIP message.

Table 8. Common Header Fields in Outbound SIP Messages

SIP header field	SIP message	Global Call function to set / send message
Accept	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Accept-Encoding	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Accept-Language	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Allow	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Call-ID	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Contact (display string and URI separately accessible separately using field-specific parameters)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REGISTER	gc_ReqService( )
Content-Disposition	INFO	gc_Extension( )
Content-Encoding	INFO	gc_Extension( )
Content-Length	INFO	gc_Extension( )
Content-Type	INFO	gc_Extension( )

‡ From and To header fields are not set in INVITE messages using SIP message information parameters.

Table 8. Common Header Fields in Outbound SIP Messages (Continued)

SIP header field	SIP message	Global Call function to set / send message
Diversion (URI separately accessible via field-specific parameter)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
Event	NOTIFY, SUBSCRIBE	gc_Extension( )
Expires	SUBSCRIBE	gc_Extension( )
From (display string separately accessible via field-specific parameter)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	REGISTER	gc_ReqService( )
Refer-To	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
Referred-By	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
Replaces	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
Request-URI	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	REGISTER	gc_ReqService( )
Require	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REGISTER	gc_ReqService( )
Supported	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REGISTER	gc_ReqService( )

‡ From and To header fields are not set in INVITE messages using SIP message information parameters.

Table 8. Common Header Fields in Outbound SIP Messages (Continued)

SIP header field	SIP message	Global Call function to set / send message
To (display string separately accessible via field-specific parameter)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	REGISTER	gc_ReqService( )
‡ From and To header fields are not set in INVITE messages using SIP message information parameters.		

Header Fields in Incoming SIP Messages

For incoming SIP messages, the Global Call library packages the header fields that the application has registered to receive as parameters in the GC_PARM_BLK that is associated with the Global Call event that notifies the application of the message. The application retrieves the parameter block by calling **gc_GetMetaEvent( )**, and can then extract the contents of the various header fields from the GC_PARM_BLK. The application must complete the retrieval of the necessary SIP message header information (for example, by copying it into its own buffer) before the next call to **gc_GetMetaEvent( )**, since the parameter block is no longer available from the metaevent buffer once the next **gc_GetMetaEvent( )** call is issued.

In addition to the header fields that the application specifically registers to receive, the GC_PARM_BLK for a message-related Global Call event may contain one or more of the header-specific parameters that were used in the previous header access methodology. It is important to note that these parameters are limited to a 255 byte data length and may potentially contain a truncation of the a header field's contents.

Table 9 lists some common SIP header fields along with the SIP message that commonly contains them and the Global Call event that is used to convey the message information to the application.

Note: The From URI and To URI in incoming INVITE messages are accessible using the **gc_GetCallInfo( )** function; see Section 7.3.9, "gc_GetCallInfo( ) Variances for IP", on page 303, for more information. In all other cases, applications must access the complete From and To header fields in order to access the URIs.

Table 9. Common Header Fields in Inbound SIP Messages

SIP header	SIP message	Global Call event
Accept	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Accept-Encoding	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Accept-Language	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
† Header field also accessible via field-specific parameter define. ‡ From and To header fields are not retrieved from INVITE messages using SIP message information parameters.		

Table 9. Common Header Fields in Inbound SIP Messages (Continued)

SIP header	SIP message	Global Call event
Allow	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Call-ID †	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Contact (display string and URI separately returned in field-specific parameters)	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled
	3xx to 6xx responses	GCEV_DISCONNECTED
Content-Disposition †	INFO	GC_CALLINFO
Content-Encoding †	INFO	GC_CALLINFO
Content-Length †	INFO	GC_CALLINFO
Content-Type †	INFO	GC_CALLINFO
Diversion (URI separately returned in field-specific parameter)	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
Event †	NOTIFY, SUBSCRIBE	GCEV_EXTENSION
Expires †	SUBSCRIBE	GCEV_EXTENSION
From ‡ (display string and full header also returned in header-specific parameters)	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled
Referred-By †	INVITE	GCEV_OFFERED
	REFER	GCEV_REQ_XFER if call transfer is enabled
Replaces †	INVITE	GCEV_OFFERED
	REFER	GCEV_REQ_XFER if call transfer is enabled
Request-URI †	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled
Require	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled

† Header field also accessible via field-specific parameter define.
‡ From and To header fields are not retrieved from INVITE messages using SIP message information parameters.

Table 9. Common Header Fields in Inbound SIP Messages (Continued)

SIP header	SIP message	Global Call event
Supported	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
To ‡ (display string and full header also returned in header-specific parameters)	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled

† Header field also accessible via field-specific parameter define.
‡ From and To header fields are not retrieved from INVITE messages using SIP message information parameters.

API Functions for Long Header Values

Because some SIP header fields (particularly those that allow multiple values to be contained in a single header field in a comma-delimited list) can be arbitrarily long, the Global Call IP library has been extended to remove the inherent 255 byte data length limitation for parameters that are contained in a GC_PARM_BLK data structure.

When using the IPSET_SIP_MSGINFO/IPPARM_SIP_HDR parameter, and the new, extended **gc_util_...** utility functions (see Section 7.2, "IP-Specific Global Call Functions", on page 278, for complete information on these functions), the maximum length of the parameter value is set by a configuration parameter that is set when the virtual board is started. Applications *must not* make any attempt to access the parameter block data directly; instead, the new, extended **gc_util_...** utility functions, which handle the extended-length data properly, should *always* be used.

The new, extended **gc_util_...** utility functions are backwards compatible and can be used with any GC_PARM_BLOCK regardless of whether it contains parameters that may exceed 255 bytes. For this reason, it is recommended that the extended functions should always be used in application code that accesses SIP header fields.

Field-Specific Parameters for SIP Header Access

Certain standard SIP header fields can be accessed using header-specific Global Call parameter IDs instead of the generic IPSET_SIP_MSGINFO / IPPARM_SIP_HDR parameter that is described in above.

The use of the header-specific parameter IDs has the following limitations:

- This mechanism is being deprecated. The defines will remain in the IP Call Control library for backward compatibility, but no further development will be done on these parameters and no issues or problems will be fixed.
- The parameter data associated with header-specific parameter IDs (that is, the header field contents) is limited to 255 bytes. You **must** use the generic IPPARM_SIP_HDR parameter ID rather than a header-specific parameter ID to handle any header field that is longer than 255 bytes.

Table 10 lists the SIP header fields that have field-specific parameter IDs, all of which are deprecated. The table also indicates the size defines that correspond to each parameter ID, each of which is equated to 255. Note that some of these parameters provide access to specific portions of the corresponding header field, such as only the URI or only the display string.

Note that there is no advantage to using the field-specific parameters that identify complete fields when setting SIP headers. Parameters that access only a part of the corresponding header field (i.e., just the URI or just the display string) may provide some convenience but should be used with caution because all of these parameter IDs are being deprecated.

When a SIP message is received, the associated parm block contained in the event metadata contains an element that uses the header-specific parameter ID for each corresponding header field that is present in the message, regardless of whether the same field is registered to be received using the generic IPSET_SIP_MSGINFO / IPPARM_SIP_HDR parameter

Table 10. Field-Specific Parameters (Deprecated) for SIP Header Access

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
Call-ID ††	IPSET_SIP_MSGINFO • IPPARM_CALLID_HDR	IP_CALLID_HDR_MAXLEN
Contact display string	IPSET_SIP_MSGINFO • IPPARM_CONTACT_DISPLAY	IP_CONTACT_DISPLAY_MAXLEN
Contact URI	IPSET_SIP_MSGINFO • IPPARM_CONTACT_URI	IP_CONTACT_URI_MAXLEN
Content-Disposition	IPSET_SIP_MSGINFO • IPPARM_CONTENT_DISPOSITION	IP_CONTENT_DISPOSITION_ MAXLEN
Content-Encoding	IPSET_SIP_MSGINFO • IPPARM_CONTENT_ENCODING	IP_CONTENT_ENCODING_MAXLEN
Content-Length	IPSET_SIP_MSGINFO • IPPARM_CONTENT_LENGTH	IP_CONTENT_LENGTH_MAXLEN
Content-Type	IPSET_SIP_MSGINFO • IPPARM_CONTENT_TYPE	IP_CONTENT_TYPE_MAXLEN
Diversion URI	IPSET_SIP_MSGINFO • IPPARM_DIVERSION_URI	IP_DIVERSION_MAXLEN
Event	IPSET_SIP_MSGINFO • IPPARM_EVENT_HDR	IP_EVENT_HDR_MAXLN
Expires	IPSET_SIP_MSGINFO • IPPARM_EXPIRES_HDR	IP_EXPIRES_HDR_MAXLEN
From display string	IPSET_SIP_MSGINFO • IPPARM_FROM_DISPLAY	IP_FROM_DISPLAY_MAXLEN

† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define.
†† Directly setting the Call-ID header field using this parameter overrides any Call-ID value that is set using the IPSET_CALLINFO / IPPARM_CALLID parameter.
‡ The Refer-To header field can only be set; it cannot be read.

Table 10. Field-Specific Parameters (Deprecated) for SIP Header Access (Continued)

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
From (complete header field, with display string, URI, and parameters)	IPSET_SIP_MSGINFO • IPPARM_FROM	IP_FROM_MAXLEN
Refer-To ‡	IPSET_SIP_MSGINFO • IPPARM_REFER_TO	IP_REFER_TO_MAXLEN
Referred-By	IPSET_SIP_MSGINFO • IPPARM_REFERRED_BY	IP_REFERRED_BY_MAXLEN
Replaces (parameter in Refer-To header field for attended call transfers)	IPSET_SIP_MSGINFO • IPPARM_REPLACES	IP_REPLACES_MAXLEN
Request-URI	IPSET_SIP_MSGINFO • IPPARM_REQUEST_URI	IP_REQURI_MAXLEN
To display string	IPSET_SIP_MSGINFO • IPPARM_TO_DISPLAY	IP_TO_DISPLAY_MAXLEN
To (complete header field, with display string, URI, and parameters)	IPSET_SIP_MSGINFO • IPPARM_TO	IP_TO_MAXLEN

† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define.
†† Directly setting the Call-ID header field using this parameter overrides any Call-ID value that is set using the IPSET_CALLINFO / IPPARM_CALLID parameter.
‡ The Refer-To header field can only be set; it cannot be read.

4.6.2 Enabling Access to SIP Header Information

The ability to set and retrieve information from SIP message header fields is an optional feature that can be enabled or disabled at the time the **gc_Start( )** function is called.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** utility functions, which must be called before the **gc_Start( )** function, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The default value of the sip_msginfo_mask field in the IP_VIRTBOARD structure disables application access to all SIP message header fields. The value IP_SIP_MSGINFO_ENABLE (possibly OR'ed with other defined mask values) must be set into the sip_msginfo_mask field for each IPT board device on which the feature is to be enabled. The following code snippet provides an example for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
```

Note: Setting value of IP_SIP_MSGINFO_ENABLE into the sip_msginfo_mask field enables overall set and retrieve access to SIP header fields for the virtual board. Enabling and disabling access to individual SIP header fields is **not** supported.

4.6.3 Enabling Long Header Values

The ability to set and retrieve SIP message header fields that exceeds 255 bytes in length is an optional feature that can be enabled at the time the **gc_Start( )** function is called.

The **INIT_IPCCLIB_START_DATA( )** utility functions, which must be called before the **gc_Start( )** function, populates the IPCCLIB_START_DATA structure with default values. The default value of the max_parm_data_size field in the IPCCLIB_START_DATA structure sets the maximum data length for parameter data in a GC_PARM_BLK structure at 255 for backward compatibility. If the application requires the ability to send and receive SIP header fields that are longer than this default maximum length (up to a maximum of 4096 bytes), it can overwrite the default value after initializing the IPCCLIB_START_DATA but before calling **gc_Start( )**. The following code snippet provides an example of setting a maximum SIP header field length of 1024 bytes for each of for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
ipcclibstart.max_parm_data_size = 1024;  /* set maximum SIP header length to 1k */
```

4.6.4 Registering SIP Header Fields to be Retrieved

In order to receive specific SIP header fields, the application must register the field names. The registration is accomplished by constructing a GC_PARM_BLK where each element contains registration information for an individual header field to be retrieved, then calling **gc_SetConfigData( )** to set the registration list in the library. Each element in the parm block uses the IPSET_CONFIG set ID and the parameter ID IPPARM_REGISTER_SIP_HEADER, plus the header field name as the parameter value. The registration of header fields only needs to be performed once for a board device, but the application is free to set a different registration list at some other time, if desired.

When registering standard SIP header fields (that is, header fields which are defined in the IETF RFC documents), the field names must be spelled consistently so that the SIP stack can recognize the header fields properly. Be certain that the spelling matches the following list (noting that case does not matter). Note that Request-URI is handled just like a standard header field, even though it is technically distinct from true header fields.

Note: In this list, header fields that are assumed to be accessible to applications to support functionality documented in this guide are marked with a †, and fields that are accessible in part or in whole via deprecated header-specific parameter defines are marked with an *.

- Accept †
- Accept-Encoding †
- Accept-Language †
- Allow †
- Allow-Events
- Authentication
- Authentication-Info

- Authorization
- Call-ID † *
- Contact † *
- Content-Disposition † *
- Content-Encoding † *
- Content-Language † *
- CSeq
- Date
- Diversion † *
- Event † *
- Expires † *
- From † *
- Max-Forwards
- Min-Expires
- Min-SE
- Proxy-Authenticate
- Proxy-Authorization
- RAck
- Referred-By † *
- Refer-To
- Replaces † *
- Request-URI † *
- Require †
- Retry-After
- Route
- RSeq
- Session-Expires
- Subscription-State
- Supported †
- To † *
- Unsupported
- Via
- Warning
- WWW-Authenticate †

The following code snippet illustrates how an application would register to receive the six SIP header fields required for use of OPTIONS messages that are not accessible via header-specific parameter defines.

Note: This example uses **gc_util_insert_parm_ref( )** rather than **gc_util_insert_parm_ref_ex( )** because it is known that header field name strings are short and never come close to the 255 byte data length limit.

```
// all devices are open
// register SIP headers to monitor

GC_PARM_BLKP parmblkp = NULL;

char *pAccept = "Accept";
char *pAcceptEnc = "Accept-Encoding";
char *pAcceptLang = "Accept-Language";
char *pAllow = "Allow";
char *pRequire = "Require";
char *pSupported = "Supported";

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAccept) + 1,
                        pAccept);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAcceptEnc) + 1,
                        pAcceptEnc);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAcceptLang) + 1,
                        pAcceptLang);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAllow) + 1,
                        pRemoteAllow);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pRequire) + 1,
                        pRequire);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pSupported) + 1,
                        pSupported);

long request_id = 0;
```

```
// SetConfigData
// NOTE: device handle is a handle to the board device
if (gc_SetConfigData(GCTGT_CCLIB_NETIF, boarddevh, parmblkp, 0,
                     GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC) != GC_SUCCESS)
{
   sprintf(str, "gc_SetConfigData(boarddevh=%ld) Failed registering SIP headers", boarddevh);
   printf ("%s"str);
}

gc_util_delete_parm_blk(parmblkp);
```

4.6.5 Setting SIP Header Fields for Outbound Messages

Note that it is not necessary for applications to register in advance the header field types that it will be setting (as described in Section 4.6.4, "Registering SIP Header Fields to be Retrieved", on page 138). Registration of header field names is only required when the application needs to *retrieve* those header fields from received messages.

Assuming that SIP message information access was enabled when the virtual board was started, applications set SIP message header fields by inserting the set ID/parm ID and value string for each field being set into a GC_PARM_BLK using **gc_util_insert_parm_ref_ex( )** or **gc_util_insert_parm_val( )**, and then either setting the header fields for the next message to be sent by calling the **gc_SetUserInfo( )** function or immediately sending the message by calling **gc_Extension( )** or another Global Call function that causes a SIP message to be sent.

When calling **gc_SetUserInfo( )** to preset SIP message header fields (which is only recommended when using the **gc_MakeCall( )** function), the **duration** parameter must be set to GC_SINGLECALL, and the information is not transmitted until the next Global Call function that sends a SIP message is issued. Note that the preset header fields will be sent in the next SIP message, so that the application must ensure that no other Global Call function is called before **gc_MakeCall( )**.

Calling the **gc_SetUserInfo( )** function results in the following behavior:

- SIP message header fields that are set do not take effect until **gc_MakeCall( )** or another function that transmits a SIP message is issued.
- Using the **gc_SetUserInfo( )** does not affect incoming SIP messages on the same channel.
- Any SIP message header fields that are set only affect the next Global Call function call.
- The **gc_SetUserInfo( )** function fails with GC_ERROR if the sip_msginfo_mask field in the IP_VIRTBOARD structure is not set to IP_SIP_MSGINFO_ENABLE. When **gc_ErrorInfo( )** is called in this case, the error code is IPERR_BAD_PARAM.

The **gc_Extension( )** function is typically used when sending supplementary SIP messages, such as INFO or OPTIONS. It is possible to use the **gc_SetUserInfo( )** function to set the header field before sending the message with the **gc_Extension( )** function call or other function that directly produces a SIP request (such as **gc_ReqService( )** for SIP REGISTER requests), but that approach is not recommended. This is the case because the preset header fields will be used in the very next SIP message that is sent, so the application must ensure that no other Global Call function is called before the intended function.

Refer to Table 8, “Common Header Fields in Outbound SIP Messages”, on page 131, to see the correspondence between the most common SIP header fields, the supported SIP messages in which these header fields are commonly set, and the Global Call functions that are called to transmit these messages.

Applications should use the IPSET_SIP_MSGINFO set ID and the IPPARM_SIP_HDR parameter ID when setting SIP header fields in the GC_PARM_BLK. This same set ID/parm ID pair can be used to set any settable SIP header field, whether it is a required field, an optional one, or a proprietary one. In each case, the parameter value that is inserted into the parameter block is a string that is the complete header field to be sent, starting with the header field name and including all required syntax elements and punctuation.

As permitted in RFC 3261 and other IETF standards, applications can insert multiple header fileds of the same type with different values, or can insert a single header field with multiple values in a comma-delimited string.

When an optional or proprietary header field is being set, the IP call control library and SIP stack simply pass through the header contents as specified by the application. The library and stack check for the presence of all header fields that are required for a specific SIP request or reply, and if such a required field is being set by the application, there may be some level of validation performed, as well. Further details regarding validation and error checking will be provided in future revisions of this document.

Note: Setting SIP message header information requires a detailed knowledge of the SIP protocol and its relationship to Global Call. The application has the responsibility to ensure that the correct SIP message information is set before calling the appropriate Global Call function to send the message.

Note that header-specific Global Call parameter IDs exist for some standard SIP header fields, but that there is no advantage to using the those parameters when setting SIP headers if the parameter accesses a complete header field. Parameters that access only a part of the corresponding header field (i.e., just the URI or just the display string) may provide some convenience, but this approach is not recommended because all of the header-specific parameter defines are being deprecated. Table 11 identifies the parameter IDs that provide access to partial header fields.

Table 11. Parameter IDs for Partial Header Field Access (Deprecated)

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
Contact display string	IPSET_SIP_MSGINFO • IPPARM_CONTACT_DISPLAY	IP_CONTACT_DISPLAY_MAXLEN
Contact URI	IPSET_SIP_MSGINFO • IPPARM_CONTACT_URI	IP_CONTACT_URI_MAXLEN
Diversion URI	IPSET_SIP_MSGINFO • IPPARM_DIVERSION_URI	IP_DIVERSION_MAXLEN
From display string	IPSET_SIP_MSGINFO • IPPARM_FROM_DISPLAY	IP_FROM_DISPLAY_MAXLEN

† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define, all of which are equated to 255.

Table 11. Parameter IDs for Partial Header Field Access (Deprecated) (Continued)

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
Replaces (parameter in Refer-To header field for attended call transfers)	IPSET_SIP_MSGINFO • IPPARM_REPLACES	IP_REPLACES_MAXLEN
To display string	IPSET_SIP_MSGINFO • IPPARM_TO_DISPLAY	IP_TO_DISPLAY_MAXLEN

† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define, all of which are equated to 255.

The following code snippet shows how to set the Request-URI header information before issuing **gc_MakeCall( )**. This translates to a SIP INVITE message with the specified Request-URI.

```
#include "gclib.h"
..
..
GC_PARM_BLK  *pParmBlock = NULL;
char         *pDestAddrBlk = "1111@127.0.0.1\0";
char         *pReqURI = "sip:2222@127.0.0.1\0";


/* Insert SIP Request-URI */
/* Add 1 to strlen for the NULL termination character */
gc_util_insert_parm_ref_ex(&pParmBlock,
                           IPSET_SIP_MSGINFO,
                           IPPARM_REQUEST_URI,
                           strlen(pReqURI) + 1,
                           pReqURI);

/* Set Call Information */
gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_SINGLECALL);

gc_util_delete_parm_blk(pParmBlock);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

/* calling the function with the MAKECALL_BLK,
the INVITE "To" field will be: 1111@127.0.0.1
the INVITE RequestURI will be: sip:2222@127.0.0.1
*/
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout, EV_ASYNC);
```

The following code snippet illustrates how an application can set a proprietary header called Remote-Party-ID. This header is a CableLabs (DCS Group) sponsored extension to transmit trusted Caller Identity and Privacy ISUP indications which have not been standardized for translation across SIP networks.

```
GC_PARM_BLKP parmblkp = NULL;
char *pRemotePartyIdHeader = "Remote-Party-ID:Alice";

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           strlen(pRemotePartyIdHeader) + 1,
                           pRemotePartyIdHeader);

gc_SetUserInfo(GCTGT_GCLIB_CRN, crn, parmblkp, GC_SINGLECALL);
```

```
gc_util_delete_parm_blk(parmblkp);

// transmit SIP message to network
...
...
```

4.6.6 Retrieving SIP Message Header Fields

The reception of most SIP requests and replies is reported to the application by means of a Global Call event, with information about the type of message contained in the metaevent data. If SIP message information access was enabled when the virtual board was started (see Section 4.6.2, "Enabling Access to SIP Header Information", on page 137), the metaevent will also contain information from SIP header fields. The application processes the Global Call event using the **gc_GetMetaEvent( )** function, and then processes the CG_PARM_BLK using Global Call utility functions.to retrieve the message type information and individual SIP header fields of interest.

Note: The application must retrieve the necessary SIP message header field information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information no longer available from the metaevent buffer.

Refer to Table 9, "Common Header Fields in Inbound SIP Messages", on page 133, to see the correspondence between SIP message type and Global Call event type for common SIP header fields.

If the application has registered one or more SIP header fields to be received (as described in Section 4.6.4, "Registering SIP Header Fields to be Retrieved", on page 138), the GC_PARM_BLK contains a separate parameter element for each registered field that was present in the received message. Each of these elements contains the IPSET_SIP_MSGINFO set ID and the IPPARM_SIP_HDR parameter ID. The associated data buffer contains the entire header field, complete with name, value, and any optional parameters. It is the application's responsibility to parse the data to determine the type of the header field.

If the received message contains multiple header field rows with the same field name, there will be a corresponding multiple set of parameter elements in the GC_PARM_BLK in the same order in which the multiple rows were arranged in the message header. If any header field contains multiple values as a comma-delimited list, it is the application's responsibility to parse the retrieved list and extract the separate values, as appropriate

The following code snippet illustrates how an application retreives registered SIP header fields when a Global Call event has been received. The example assumes that the header field name has been registered and that the event has already been received.

```
char siphdr[IP_SIP_HDR_MAXLEN];
GC_PARM_DATA_EXT parm_data
INIT_GC_PARM_DATA_EXT(&parm_data)

while ((ret = gc_util_next_parm_ex(pParmBlock, &parm_data)) == GC_SUCCESS)
{
   switch (parm_data.parm_ID)
   {
      case IPPARM_SIP_HDR:
         strncpy(siphdr, (char*)parm_data.pData, parm_data.data_size);
         siphdr[parm_data.data_size]='\0';
         sprintf(m_DisplayString, "\t\tGeneric Sip Header = %s", siphdr);
         printf("%s", m_DisplayString);
         break;
   }
}
```

In addition to the IPPARM_SIP_HDR elements that correspond to the registered header fields, the parm block will also contain elements that use the deprecated field-specific parameter IDs listed in Table 10, "Field-Specific Parameters (Deprecated) for SIP Header Access", on page 136. Some of these field-specific parameters provide access to a specific part of the corresponding header field (specifically just the display string or just the URI) rather than the complete header field.

The following code demonstrates how to copy the Request-URI from a GCEV_OFFERED event using the (deprecated) field-specific parameter ID IPPARM_REQUEST_URI. The GC_PARM_BLK structure containing the data is referenced via the extevtdatap pointer in the METAEVENT structure. In this particular scenario, the GCEV_OFFERED event is generated as a result of receiving an INVITE message.

```
#include "gclib.h"
..
..
METAEVENT  metaevt;
GC_PARM_BLK   *pParmBlock = NULL;
GC_PARM_DATA  *parmp = NULL;
char          reqestURI[IP_REQUEST_URI_MAXLEN];

/* Get Meta Event */
gc_GetMetaEvent(&metaevt);

switch(metaevt->evttype)
   {
    .
    .
    .
   case GCEV_OFFERED:
      currentCRN = metaevt->crn;
      pParmBlock = (GC_PARM_BLK*)(metaevt->extevtdatap);
      parmp = NULL;

      /* going thru each parameter block data*/
      while ((parmp = gc_util_next_parm_ex(pParmBlock,parmp)) != 0)
      {
         switch (parmp->set_ID)
         {
         /* Handle SIP message information */
            case IPSET_SIP_MSGINFO:
               switch (parmp->parm_ID)
               {
               /* Copy Request URI from parameter block */
               /* NOTE: value_size = string length + 1 (for the NULL termination) */
                  case IPPARM_REQUEST_URI:
                     strncpy(requestURI, parmp->value_buf, parmp->value_size);
                     break;
               }
         }
      break;
      }
    .
    .
    .
   }
```

4.7 Using MIME Bodies in SIP Messages (SIP-T)

When using SIP, the Global Call library supports the sending and receiving of messages that include a single-part or multipart MIME body.

This feature was implemented primarily to allow applications to send and receive SIP Telephony (SIP-T) information, which is encoded in a MIME message body as defined in RFC 3372, a document which describes a framework for SIP-PSTN interworking gateways. This capability allows the encapsulation of ISUP in the SIP body during or after call setup, and the use of the INFO method for mid-call signaling. With the use of a separate SS7 signaling stack to translate the ISUP information, applications can route SIP messages with dependencies on ISUP to provide ISUP transparency across SS7-ISUP internetworking.

The Global Call implementation of SIP MIME messages is very general, so that it should support MIME for a variety of other purposes besides SIP-T, such as text messaging. The call control library only copies data to and from a SIP MIME body. With the exception of SDP (Session Description Protocol), the Global Call library treats MIME body information as raw data and does not parse or translate information that is encapsulated in SIP MIME messages. (SDP is not exposed to the application like other MIME-encoded data because the call control library controls media negotiations internally.)

4.7.1 SIP MIME Overview

The Global Call library handles single-part MIME and multipart MIME in the same way to simplify application coding. The library uses two levels of GC_PARM_BLK data structures to contain information being embedded into or extracted from MIME messages. The top-level GC_PARM_BLK structure contains a list of one or more lower-level GC_PARM_BLK structures that contain the header and body information for each MIME part. When an application sends a single MIME part in a SIP message that already includes a MIME part for SDP (which is not exposed to applications), the library transparently creates a multipart MIME message with the appropriate multipart headers. In the case where an incoming message has multipart MIME embedded in a multipart MIME part (nested parts), the Global Call library parses through all the parts in order and extracts them to a flat list of data structures.

For incoming SIP messages with MIME information, the call control library creates a Global Call event corresponding to the message type with GC_PARM_BLK structures attached. Standard Global Call practices are used to retrieve the GC_PARM_BLKs, and all information in each MIME part is accessed through parameters in the corresponding GC_PARM_BLK structure.

For outgoing SIP messages, the application must populate GC_PARM_BLK structures with parameters that specify the content of all the MIME parts to be sent, and then set the MIME information before or at the time of calling the relevant Global Call function that sends the SIP message.

Figure 40 shows the relationships between Global Call function calls, SIP messages, and Global Call events for outgoing and incoming SIP messages with MIME content in a normal call setup/teardown scenario. Figure 41 shows the same relationships in a reject scenario.

Figure 40. SIP MIME Scenario for Normal Call Setup and Teardown

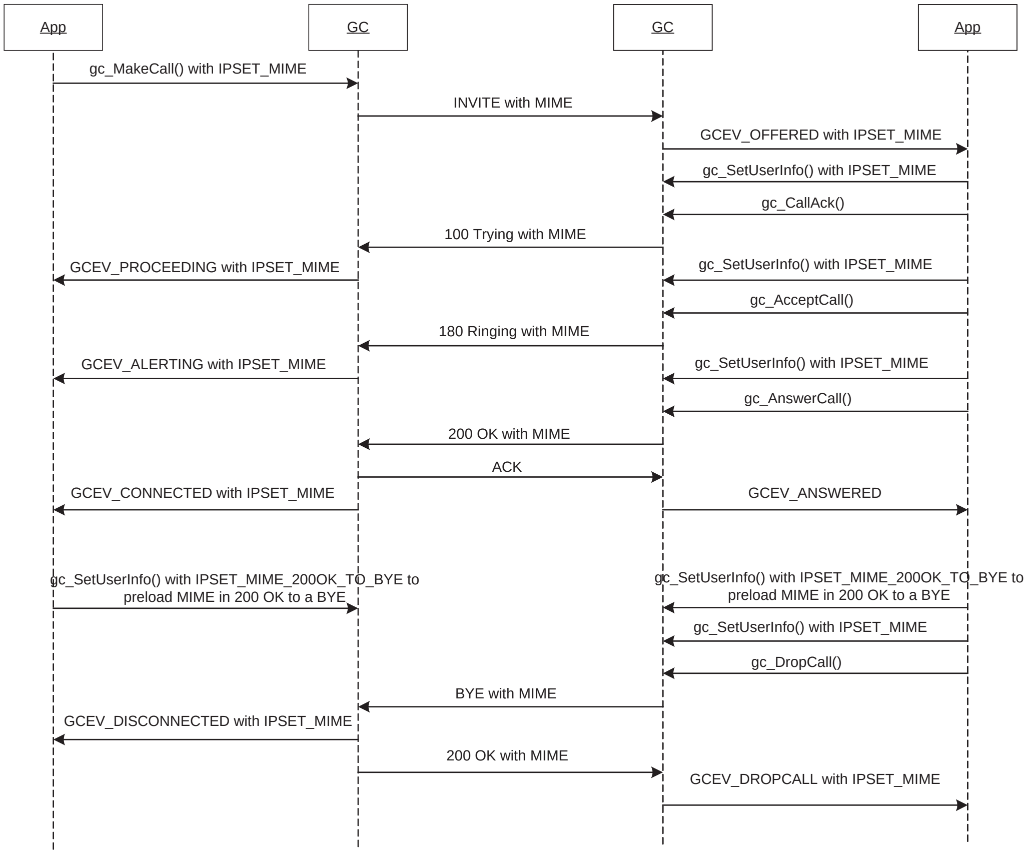
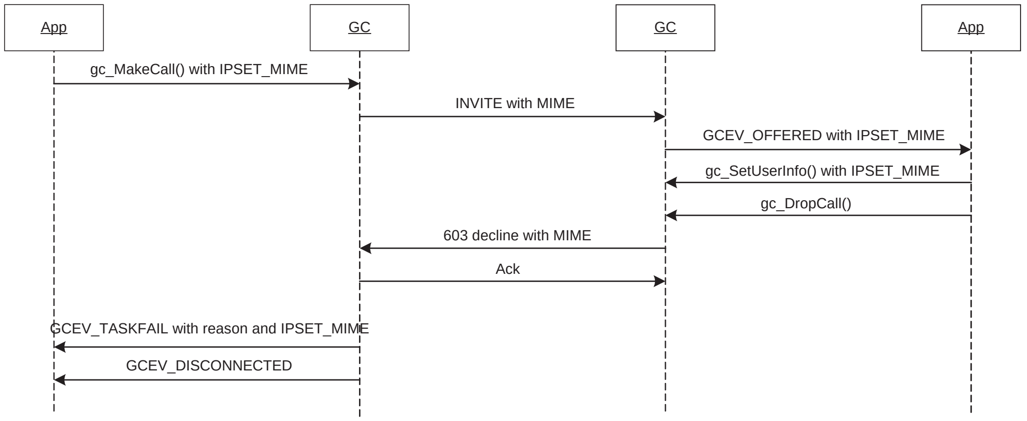
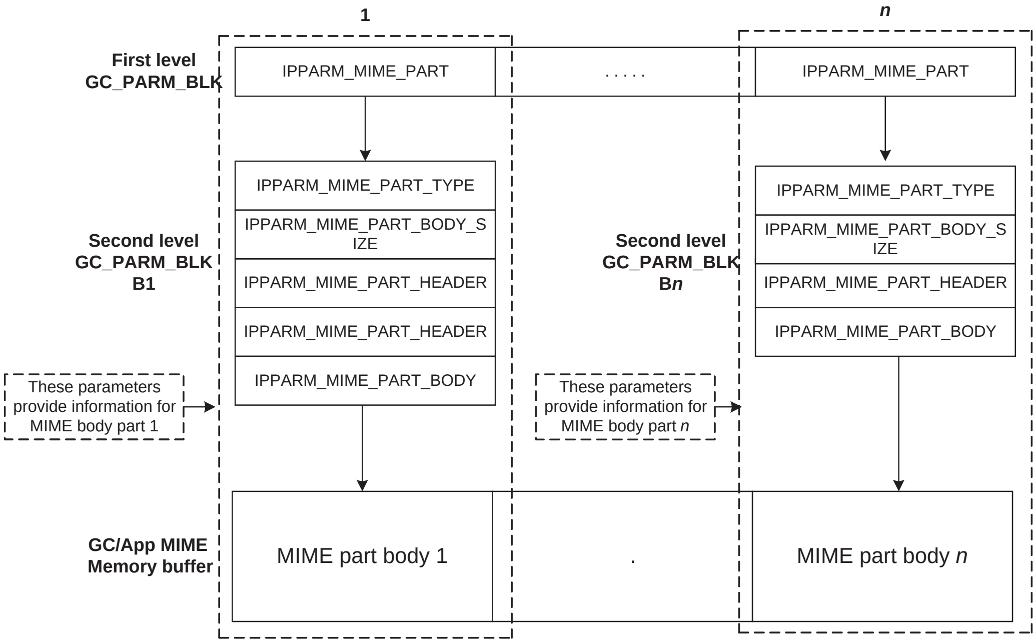

App
GC
GC
App
gc_MakeCall() with IPSET_MIME
INVITE with MIME
GCEV_OFFERED with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_CallAck()
100 Trying with MIME
GCEV_PROCEEDING with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_AcceptCall()
180 Ringing with MIME
GCEV_ALERTING with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_AnswerCall()
200 OK with MIME
ACK
GCEV_CONNECTED with IPSET_MIME
GCEV_ANSWERED
gc_SetUserInfo() with IPSET_MIME_200OK_TO_BYE to preload MIME in 200 OK to a BYE
gc_SetUserInfo() with IPSET_MIME_200OK_TO_BYE to preload MIME in 200 OK to a BYE
gc_SetUserInfo() with IPSET_MIME
gc_DropCall()
BYE with MIME
GCEV_DISCONNECTED with IPSET_MIME
200 OK with MIME
GCEV_DROPCALL with IPSET_MIME

Figure 41. SIP MIME Scenario for Rejected Call


App
GC
GC
App
gc_MakeCall() with IPSET_MIME
INVITE with MIME
GCEV_OFFERED with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_DropCall()
603 decline with MIME
Ack
GCEV_TASKFAIL with reason and IPSET_MIME
GCEV_DISCONNECTED

Global Call uses two levels of GC_PARM_BLK data structures to handle MIME parts. The top-level GC_PARM_BLK contains the parameter set ID IPSET_MIME and one or more IPPARM_MIME_PART parameters, each of which points to a second-level GC_PARM_BLK structure that contains parameters for a specific MIME part. Within the second-level structure are three mandatory parameters that identify the type, size, and body data buffer location for the MIME part, plus an optional and possibly multiple parameter for MIME part header lines.

Figure 42. SIP MIME GC_PARM_BLK Structure



4.7.2 Enabling and Configuring the SIP MIME Feature

SIP MIME is a feature that can be disabled or enabled at the time the **gc_Start( )** function is called.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions, which must be called before the **gc_Start( )** function, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The default value of the sip_msginfo_mask field in the IP_VIRTBOARD structure disables access to SIP message information fields (headers) and the SIP MIME feature. The default sip_msginfo_mask field value must be overridden with the value IP_SIP_MIME_ENABLE for each IPT board device on which SIP MIME capabilities are to be enabled. The following code snippet provides an example for two virtual boards:

```
.
.
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MIME_ENABLE; /* override SIP MIME default */
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MIME_ENABLE; /* override SIP MIME default */
.
.
```

When the SIP MIME feature is enabled, a dedicated MIME memory pool is allocated by the Global Call library at initialization time, according to data that is contained in the MIME_MEM data structure that is in IP_VIRTBOARD. Because the size of a MIME body is potentially unlimited, the application is in the best position to set the size and number of memory buffers in the pool by overriding the default values in the MIME_MEM structure.

The buffer size should be big enough for each anticipated MIME part, including the MIME part body and all MIME part headers, but should not be larger than the maximum size permitted by the transport protocol. The default transport protocol, UDP over Ethernet, can handle up to 1500 bytes, so the MIME buffer size should be no more than 1500 if using UDP. The default buffer size value that is set by the INIT_IP_VIRTBOARD( ) function is 1500.

The number of buffers should be large enough to handle SIP-T on all channels in both incoming and outgoing directions. To allow two buffers per direction plus one additional buffer for preloading the MIME information for the 200OK to BYE message that is sent automatically when BYE is received, the default number of buffers is 5 times the value of sip_max_calls.

Note that the MIME memory pool is completely separate from the application memory pool, and that it is only allocated if SIP MIME is enabled when the virtual board is initialized.

4.7.3 Getting MIME Information

In this section, we will consider the following SIP message as an example:

```
INVITE sip:user2@127.0.0.1 SIP/2.0
From: <sip:user1@127.0.0.1>;tag=0-13c4-3f9fecfb-1a356266-56c9
To: <sip:user2@127.0.0.1>
Call-ID: 93d5f4-0-13c4-3f9fecfb-1a356266-2693@127.0.0.1
CSeq: 1 INVITE
Via: SIP/2.0/UDP 146.152.84.141:5060;received=127.0.0.1;branch=z9hG4bK-3f9fecfb-
1a356270-61ce
Max-Forwards: 70
Supported: 100rel
Mime-Version: 1.0
Contact: <sip:user1@127.0.0.1>
Content-Type: multipart/mixed ;boundary=unique-boundary-1
Content-Length: 886

--unique-boundary-1
Content-Type: application/SDP ;charset=ISO-10646

v=0
o=jpeterson 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP seminar
c=IN IP4 MG122.level3.com
t=2873397496 2873404696
m=audio 9092 RTP/AVP 0 3 4

--unique-boundary-1
Content-Type: application/ISUP ;version=nxv3 ;base=etsi121
Content-Disposition: signal ;handling=optional
Content-User: Intel ;type=demo1

01 00 49 00 00 03 02 00 07 04 10 00 33 63 21
43 00 00 03 06 0d 03 80 90 a2 07 03 10 03 63
53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b
0e 95 1e 1e 1e 06 26 05 0d f5 01 06 10 04 00
```

```
--unique-boundary-1—
Content-Type: image/jpeg
Content-Transfer-Encoding: base64

iQCVAwUBMJrRF2N9oWBghPDJAQE9UQQAtl7LuRVndBjrk4EqYBIb3h5QXIX/LC//
jJV5bNvkZIGPIcEmI5iFd9boEgvpirHtIREEqLQRkYNoBActFBZmh9GC3C041WGq
uMbrbxc+nIs1TIKlA08rVi9ig/2Yh7LFrK5Ein57U/W72vgSxLhe/zhdfolT9Brn
HOxEa44b+EI=
--unique-boundary-1—
```

Note that this example of a SIP MIME message includes three MIME parts, one of which contains SDP, which is handled internally by the Global Call library (except for the special case of responses to OPTIONS requests). When handling this message, the application sees only two MIME parts because SDP is not exposed to applications.

Also note that this example illustrates a SIP INVITE message, which is only one of many different SIP message types that can contain MIME parts intheir bodies.

Table 12. Global Call Events for Incoming SIP Messages that can Contain MIME Bodies

Incoming SIP Message	Global Call Event
BYE	GCEV_DISCONNECTED
INFO	GCEV_CALLINFO
INVITE	GCEV_OFFERED
NOTIFY	GCEV_EXTENSION
OPTIONS	GCEV_EXTENSION
REFER	GCEV_REQ_XFER
SUBSCRIBE	GCEV_EXTENSION
100 Trying	GCEV_PROCEEDING
180 Ringing	GCEV_ALERTING
200 OK to BYE	GCEV_DROPCALL
200 OK to INVITE	GCEV_CONNECTED
3xx to 6xx Request Failure	GCEV_DISCONNECTED

When receiving a Global Call event with an attached GC_PARM_BLK that contains the parameter IPPARM_MIME_PART, the application needs to retrieve the pointer to the second-level GC_PARM_BLK from the value of IPPARM_MIME_PART. In this example, there are three MIME parts in the message, but only two IPPARM_MIME_PART parameters in the GC_PARM_BLK because the SDP MIME part is not exposed. The order of the IPPARM_MIME_PART parameters is the same as the order of the MIME parts in the SIP message.

The first-level GC_PARM_BLK contains the following parameters and values for the example shown above:

```
IPPARM_MIME_PART
    0x78339ff0
```

    [address of first second-level GC_PARM_BLK (B1)]

`IPPARM_MIME_PART` (required)
`0x78356144`
[address of second second-level GC_PARM_BLK (B2)]

The first second-level GC_PARM_BLK (B1), at address 0x78339ff0 in this example, contains the following parameters and values, which represent the information for the first non-SDP MIME part in the example shown above:

`IPPARM_MIME_PART_TYPE`
`Content-Type: application/ISUP ;version=nxv3 ;base=etsi121`
[data from MIME part header in received MIME message]

`IPPARM_MIME_PART_BODY_SIZE`
`182`
[size of received data in buffer]

`IPPARM_MIME_PART_BODY`
`0x329823e8`
[address of buffer]

`IPPARM_MIME_BODY_HEADER` [optional parameter]
`Content-Disposition: signal ;handling=optional`
[data from MIME part header in received MIME message]

`IPPARM_MIME_BODY_HEADER` [optional parameter]
`Content-User: Intel ;type=demo1`
[data from MIME part header in received MIME message]

The buffer at the address given in the value of IPPARM_MIME_PART_BODY (0x329823e8 in this example) contains the data that was received in the MIME part body:

```
01 00 49 00 00 03 02 00 07 04 10 00 33 63 21
43 00 00 03 06 0d 03 80 90 a2 07 03 10 03 63
53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b
0e 95 1e 1e 1e 06 26 05 0d f5 01 06 10 04 00
```

The second, second-level GC_PARM_BLK (B2), at address 0x78356144 in this example, contains the following parameters and values, which represent the information for the second non-SDP MIME part in the example shown above:

`IPPARM_MIME_PART_TYPE`
`Content-Type: image/jpeg`
[data from MIME part header in received MIME message]

`IPPARM_MIME_PART_BODY_SIZE`
`208`
[size of received data in buffer]

`IPPARM_MIME_PART_BODY`
`0x3298a224`
[address of buffer]

`IPPARM_MIME_BODY_HEADER` [optional parameter]
`Content-Transfer-Encoding: base64`
[data from MIME part header in received MIME message]

The buffer at the address given in the value of IPPARM_MIME_PART_BODY (0x3298a224 in this example) contains the data that was received in the MIME part body:

```
iQCVAwUBMJrRF2N9oWBghPDJAQE9UQQAtl7LuRVndBjrk4EqYBIb3h5QXIX/LC//
jJV5bNvkZIGPIcEmI5iFd9boEgvpirHtIREEqLQRkYNoBActFBZmh9GC3C041WGq
uMbrbxc+nIs1TIKlA08rVi9ig/2Yh7LFrK5Ein57U/W72vgSxLhe/zhdfolT9Brn
HOxEa44b+EI=
```

Note that the data that is retrieved from each MIME part body is copied into the buffer as a continuous block of binary data whose length (in bytes) is indicated in IPPARM_MIME_PART_BODY_SIZE. No type checking or data formatting is performed by the Global Call library. Note that a MIME part body does not necessarily end with '\0', and that a MIME body might contain '\0' as part of the body itself.

All GC_PARM_BLK structures (on both levels) and MIME part body buffers will be freed when the next Global Call event is accessed. The application must therefore copy the necessary parameters and the data buffers before processing the next Global Call event.

Code Example

The following code example illustrates the retrieval of MIME information from a GCEV_OFFERED event. It prints out every MIME part header and MIME part body (except for any SDP) that exists in the SIP INVITE message.

```
INT32 processEvtHandler()
{
   METAEVENTmetaEvent;
   GC_PARM_BLK*parmblkp = NULL;
   GC_PARM_DATAPt_gcParmDatap = NULL;
   GC_PARM_BLK*parmblkp2 = NULL;
   :
   switch (evtType)
   {

   case GCEV_OFFERED:
      /* received GC event, parse PARM_BLK examine
       * extension data
       */
      parmblkp = (GC_PARM_BLK *) metaEvent.extevtdatap;
      while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
      {
         switch(t_gcParmDatap->set_ID)
         {
         case IPSET_MIME:
            switch(t_gcParmDatap->parm_ID)
            {
            case IPPARM_MIME_PART:
               /* Get MIME part pointer */
               parmblkp2= (GC_PARM_BLK*)(*(UINT32*)( t_gcParmDatap ->value_buf));

               if(NULL == parmblkp2 || 0 != getMIMEPart(parmblkp2))
               {
                  printf("\n!!!error getting MIME part!!!\n");
                  return -1;
               }
               break;
            }
            break;
         }
      }
   }
   :
}



INT32 getMIMEPart(GC_PARM_BLK* parmblkp)
{
   GC_ARM_DATAP     t_gcParmDatap = NULL;
   char             temp[256];
   UINT32           bodySize = 0;
   char             *appBuff = NULL;
   char             *bodyBuff = NULL;

   /* get the first param data*/
   t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap);

   /* Get MIME type info, this has to be the first parameter */
   if(IPSET_MIME == t_gcParmDatap->set_ID && IPPARM_MIME_PART_TYPE == t_gcParmDatap->parm_ID)
   {
       strncpy(temp, (char*)t_gcParmDatap->value_buf, t_gcParmDatap->value_size);
       printf("\t Content-Type = %s\n", temp);
   }
   else
   {
      /* error condition */
      printf("\n !!! first parameter in MIME part is not MIME type!!!\n");
      return -1;
   }

   /* Get the rest of MIME info*/
   while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
   {
      switch(t_gcParmDatap->set_ID)
      {
         case IPSET_MIME:
            switch(t_gcParmDatap->parm_ID)
            {
               case IPPARM_MIME_PART_TYPE:
                  /* duplicate MIME part, error out */
                  printf("\n!!!Duplicate MIME part error!!!\n");
                  return -1;
                  break;

               case IPPARM_MIME_PART_BODY_SIZE:
                  /* Get MIME part body size */
                  bodySize = *(UINT32*)(t_gcParmDatap->value_buf);
                  printf("\t MIME part body Size = %d\n", bodySize);
                  break;

               case IPPARM_MIME_PART_HEADER:
                  /* Get MIME part header */
                  strncpy(temp, (char*)t_gcParmDatap->value_buf, t_gcParmDatap->value_size);
                  printf("\t MIME part header = %s\n", temp);
                  break;

               case IPPARM_MIME_PART_BODY:
                  /* get body buffer pointer */
                  bodyBuff = (char*)(*(UINT32*)(t_gcParmDatap->value_buf));

                  /* copy MIME part body */
                  if(bodySize>0)
                  {
                     /* allocate memory */
                     AppBuff = (char*)malloc(bodySize+1);
                     memcpy(appBuff, bodyBuff, bodySize);
                  }
                  else
                  {
                     /*error body size must be available*/
                     printf("\n!!! Body Size not available error !!!\n");
                     return -1;
                  }
                  /* Null terminated */
                  appBuff[bodySize] = '\0';

                  /* Only print the buffer content as string */
                  /* For binary data the buffer is not printable*/
                  printf("\t MIME part Body:\n%s\n",appBuff);

                  /* Free allocated memory*/
                  free(appBuff);
                  break;

            }
            break;

      }
   }

   :
   return 0;
}
```

4.7.4 Sending MIME Information

Table 13 lists the Global Call functions that can be used to send SIP messages with MIME information using the IPSET _MIME parameter set ID in the attached GC_PARM_BLK. Except in the cases of **gc_MakeCall( )** and **gc_Extension( )**, sending a SIP message with MIME requires two function calls, **gc_SetUserInfo( )** to set the information, and a second function to cause the library to send the message.

Table 13. Global Call Functions for SIP MIME Messages Using IPSET_MIME

Global Call Function to Set MIME Parameter Block	Global Call Function to Send MIME Message	Device Type	Outgoing SIP Message with MIME
---	**gc_MakeCall( )**	LD	INVITE
---	**gc_Extension( )**	CRN or LD	INFO, OPTIONS, SUBSCRIBE, NOTIFY
gc_SetUserInfo( )	**gc_CallAck( )**	CRN	100 Trying
gc_SetUserInfo( )	**gc_AcceptCall( )**	CRN	180 Ringing
gc_SetUserInfo( )	**gc_AnswerCall( )**	CRN	200OK to INVITE
gc_SetUserInfo( )	**gc_DropCall( )**	CRN	603 Decline if before call setup BYE if after call setup

If the application only needs to send a single MIME part but the call control library also needs to send SDP information, the firmware automatically and transparently constructs the required multipart MIME message.

If the application needs to send multipart MIME, all the MIME information is set collectively within one function call on the given device by inserting multiple IPPARM_MIME_PART parameters in the desired order to construct a multipart MIME body. The MIME information set by current function always overwrites any MIME information set by previous functions, so that an application **cannot** set multiple MIME parts by calling **gc_SetUserInfo( )** multiple times.

The parameter set ID IPSET_MIME_200OK_TO_BYE is used for a special case of MIME message. Unlike other outgoing SIP messages that are sent explicitly by Global Call functions, the 200 OK to BYE message is sent automatically when a BYE is received. In order to attach MIME information to a 200 OK to BYE message, the MIME information has to be pre-loaded by **gc_SetUserInfo**( ) with set ID IPSET_MIME_200OK_TO_BYE on a channel before the GCEV_DISCONNECTED event (SIP BYE message) is received. If a MIME message with IPSET_MIME_200OK_TO_BYE parameters is not set before the GCEV_DISCONNECTED event (BYE) is received, the automatic 200 OK message will be sent without any MIME body. Note that the parameter set ID must be set to IPSET_MIME_200OK_TO_BYE in **every** GC_PARM_BLK associated with the message, not just the top-level block. MIME information set with IPSET_MIME_200OK_TO_BYE and MIME information set with IPSET_MIME are kept independent of each other on a given channel.

The data that is to be sent in the MIME part body is copied into the message MIME part from an application buffer. The data in the buffer must match the data type that is specified by the IPPARM_MIME_PART_TYPE parameter. The Global Call library treats the buffer as a continuous block of binary data of the length (in bytes) specified in IPPARM_MIME_PART_BODY_SIZE; no type checking or formatting is performed. Note that a MIME body part does not necessarily end with '\0', and that a MIME body might contain '\0' as part of the body itself.

Constructing and setting a MIME message is a multi-part process that can be broken down into several sub-processes:

1. Create and populate a separate GC_PARM_BLK structure for each MIME part to be sent in the SIP message.
2. Create a top-level GC_PARM_BLK structure and populate it with IPPARM_MIME_PART parameters that point to the GC_PARM_BLK structures created in the first step.
3. Set or send the message by calling the appropriate Global Call function.
4. Clean up the data structures after the function returns.

Create MIME part structures

The process of constructing an outgoing SIP MIME message begins by constructing a separate GC_PARM_BLK structure for each MIME part to be sent in the message:

1. Create a GC_PARM_BLK structure.
2. Insert the required IPPARM_MIME_PART_TYPE parameter to identify the MIME part type.
3. Insert any MIME part headers via one or more optional IPPARM_MIME_PART_HEADER parameters.
4. Insert the required IPPARM_MIME_PART_BODY_SIZE parameter to identify the actual number of bytes to be copied from the application buffer to the MIME part body.

5. Insert the required IPPARM_MIME_PART_BODY parameter to point to the application buffer that contains the data for the MIME part body. Note that the Global Call library treats the buffer as a continuous block of binary data, and that the data must have the appropriate format for the MIME part type specified in the IPPARM_MIME_PART_TYPE parameter.

Create top-level GC_PARM_BLK

After repeating the preceding procedure for each MIME part to be sent in the SIP message, construct the top-level data structure that lists the MIME part structures:

1. Create a GC_PARM_BLK structure.
2. Insert a required IPPARM_MIME_PART parameter to point to the GC_PARM_BLK structure for the first MIME part in the message.
3. Repeat Step 2 for each additional MIME part, inserting the parameters in order of how the MIME parts should be organized in the message.

Set/send message data and clean up

After creating and populating the top-level GC_PARM_BLK structure that lists all the MIME parts to be sent in the SIP message, set or send the message and clean up the set-up structures:

1. Call gc_SetUserInfo( ) or gc_MakeCall( ) with a pointer to the top-level GC_PARM_BLK to set or send the MIME message data.
2. Delete all GC_PARM_BLK structures created during the set-up process after the Global Call function returns.
3. Optionally, free the application buffer holding the MIME part body data, since that data has been copied into the dedicated MIME buffer when the function was called. Or you can choose to not free the application buffer and instead reuse it for the next MIME part body.

Code Example

The following code example constructs a single part MIME message and uses the **gc_MakeCall( )** function to send it in an INVITE message.

```
#include "gclib.h"
..
..
GC_PARM_BLK *pParmBlockA = NULL;
GC_PARM_BLK *pParmBlockB = NULL;
char *pBodyType = "Content-Type: application/ISUP ;version=nxv3 ;base=etsi121";
char *pBody = "01 00 49 00 00 03 02 00 07 04 10 00 33 63 21\r\n43 00 00 03 06 0d 03 80 90 a2 07
03 10 03 63\r\n53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b\r\n0e 95 1e 1e 1e 06 26 05 0d f5 01
06 10 04 00";
char *pPartHeader1 = "Content-Disposition: signal ;handling=optional";
char *pPartHeader2 = "Content-User: Intel ;type=demo1";

/* Insert Content-Type field */
/* Add 1 to strlen for the NULL termination character */
gc_util_insert_parm_ref(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_TYPE,
                        (unsigned char)(strlen(pBodyType) + 1),
                        pBodyType);
```

```
/* Insert Body Size  */
gc_util_insert_parm_val(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_BODY_SIZE,
                        sizeof(unsigned long),
                        strlen(pBody));

/* Insert MIME part Body Pointer  */
gc_util_insert_parm_val(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_BODY,
                        sizeof(unsigned long),
                        (unsigned long)pBody);

/* Insert other header fields */
gc_util_insert_parm_ref(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_HEADER,
                        (unsigned char)(strlen(pPartHeader1) + 1),
                        pPartHeader1);

/* Insert other header fields */
gc_util_insert_parm_ref(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_HEADER,
                        (unsigned char)(strlen(pPartHeader2) + 1),
                        pPartHeader2);

/* Insert parm block B pointer to parm block A */
gc_util_insert_parm_val(&pParmBlockA,
                        IPSET_MIME,
                        IPPARM_MIME_PART,
                        sizeof(unsigned long),
                        (unsigned long)pParmBlockB);

/* Set Call Information */
gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlockA, GC_SINGLECALL);

gc_util_delete_parm_blk(pParmBlockB);
gc_util_delete_parm_blk(pParmBlockA);

.
.
.

/* Make a call */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout, EV_ASYNC);
```

4.7.5 MIME Error Conditions

When using the SIP MIME feature, any of the following conditions causes the Global Call function to return an error with the last error set to IPERR_BAD_PARAM:

- A Global Call function attempts to set MIME information when the SIP MIME feature was not enabled by setting IP_SIP_MIME_ENABLE in the IP_VIRTBOARD structure at initialization time.
- The application attempts to set MIME information with the MIME body part size larger than the MIME memory buffer size that was configured during initialization.
- The total size of MIME parts is greater than 1500 bytes when using UDP.

If the MIME memory pool is empty, or if the configured MIME buffer size is smaller than the MIME body of an incoming SIP-T message, a TASKFAIL event is sent to the application with the reason set to IPEC_MIME_POOL_EMPTY or IPEC_MIME_BUFF_TOO_SMALL, respectively. In addition, these error conditions also cause a response message with response code 486(Busy Here) to be sent to the remote UA. The current transaction will be terminated without causing the state of the current call to change.

4.8 Specifying Transport for SIP Messages

When a virtual board is configured with default values in the IP_VIRTBOARD data structure, the supported transport protocol for all SIP messages is UDP. Applications do not have the ability to send messages using TCP, and incoming TCP messages are refused.

By setting non-default parameter values in the IP_VIRTBOARD before calling **gc_Start( )**, applications can enable support of TCP as well as UDP. In addition to enabling overall TCP support, the application can configure the board to use TCP as the default transport protocol, and can set the persistence of TCP connections. See Section 4.1.2, “Configuring SIP Transport Protocol”, on page 99, for details about the configuration process.

When TCP is enabled, incoming TCP messages are accepted, and if the application needs to determine the transport protocol it can access the Request-URI in the Global Call event as described in Section 4.6.6, “Retrieving SIP Message Header Fields”, on page 144. When responding to a SIP request, the application does not need to specify TCP because the transport parameter is already be present in the Request-URI.

SIP requests that are sent by the application outside of a SIP dialog (for example, INVITE, SUBSCRIBE, or NOTIFY) normally use the default transport protocol, but the application can override the default to send a specific request using the non-default protocol by setting a “transport=” parameter in the Request-URI header field before the message is sent. If the default transport is UDP, the relevant parameter string to override the default is “;transport=tcp”; if the default transport is TCP, the relevant parameter string to override the default is “;transport=udp”. Setting the transport for a specific SIP request requires that the SIP message information access feature be enabled and uses the process described in Section 4.6.5, “Setting SIP Header Fields for Outbound Messages”, on page 141. The following code lines illustrate how a Request-URI with transport parameter would be inserted into the parameter block for the message to be sent.

```
sprintf(strReqURI, "sip:%s:%d;transport=tcp", strIPaddr, intPort);
gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_REQUEST_URI,
                        strlen(strReqURI),
                        strReqURI);
```

For SIP requests within a dialog (for example, INFO, NOTIFY, or REFER), there is no need to set the transport protocol if the persistence configuration item in IP_VIRTBOARD is set to ENUM_PERSISTENCE_TRANSACT_USER (the default value) because the existing TCP connection will be used.

BYE requests are exceptions to the general TCP behavior in several respects. First, BYE requests always make a new connection; an existing TCP connection is not used even if TCP is configured for user persistence. Second, a default transport protocol setting of TCP or a “;transport=tcp” parameter in the Request-URI header field is not sufficient to force TCP for a BYE request. Instead, it is necessary to also set “;transport=tcp” in the Contact URI header field.

4.9 Handling SIP Transport Failures

The Global Call SIP implementation provides facilities to retry a SIP request when a transport failure occurs as well as notifying the application of the failure. The retry logic used by the SIP stack is determined by the value that is set for the E_SIP_RequestRetry field in the IP_VIRTBOARD configuration structure that is used when the virtual board is started. The default configuration enables all allowable retries.

The following code snippet illustrates the general procedure for setting up the IP_VIRTBOARD structure to specify non-default request retry behavior. This specific example disables request retries following transport failure. Note that all data structure fields that are not explicitly set are assumed to contain their default values as configured by the **INIT_IP_VIRTBOARD( )** function.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// Enable SIP Message Info to allow transport selection for individual requests
virtBoards[0].ip_sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE;

//enable TCP for individual requests
virtBoards[0].E_SIP_tcpenabled = ENUM_Enabled;
virtBoards[0].E_Persistence = ENUM_PERSISTENCE_TRANSACT_USER;

//disable SIP request retry
virtboard[0].E_SIP_RequestRetry = ENUM_REQUEST_RETRY_NONE
```

When UDP is used as the transport protocol, the SIP stack automatically retries the request on the same address until a timeout occurs or a response is received. When such a timeout occurs there is generally no point in retrying further on the same address, but having the stack automatically retry on any additional addresses that are contained in the DNS server may be useful. All request retry configuration options that enable retry include this type of retry using DNS entries.

When TCP is used as the transport protocol, a request may fail because the destination is not able to accept TCP in addition to other failure causes. When a TCP request fails, it is generally desirable to have the stack retry the request using UDP, but because a UDP request is retried automatically until a response is received or the request times out, the time interval before the application receives a final fatal transport error may be significantly extended. Because of this, the options for enabling request retry allow retry using UDP on the same address for a TCP failure to be enabled separately in addition to retrying using addresses from the DNS server. Additionally, the SIP stack only retries TCP requests on the same address using UDP if the failure reason indicates that there is a reasonable possibility that the UDP request will succeed. In particular, there is little point in retrying if the failure was a 503 Service Unavailable because sending a UDP request to a busy server is no more likely to succeed than the failed TCP request. Another case where retrying a failed TCP request is not appropriate is if the failed connection was a connection to a proxy, since a failed connection to a proxy indicates that the proxy is not able to accept TCP or that the proxy is down—a fatal error in either case.

An important third case occurs when an application attempts a request using UDP, but the request is forced to TCP because of its size. In this case, the application supplies an address that is valid for UDP transport because that is the protocol it assumes will be used. If the connection fails because the destination cannot accept TCP, it is appropriate for the SIP stack to retry the same address over UDP without the application's intervention, because a UDP request is what the application expected to be sent in the first place. A separate configuration option is provided to accommodate this specific circumstance while disabling retry on the same address for requests explicitly sent over TCP.

When a request retry occurs, the Global Call IP library generates a GCEV_EXTENSION event that contains the parameter set ID IPSET_SIP_REQUEST_ERROR and the parameter ID IPPARM_SIP_DNS_CONTINUE. The associated parameter value is a simple data structure defined as follows:

```
typedef sruct {
    UINT32  Error;
    char    Method[IP_SIP_METHODSIZE];
} REQUEST_ERROR, *REQUEST_ERRORP;
```

The Error field in this structure is an enumerated error code value, and the Method array contains up to IP_SIP_METHODSIZE characters of the method name. The defined values for the Error field are:

IP_SIP_REQUEST_503_RCVD
Connection failed due to 503 Service Unavailable or other fatal error cause

IP_SIP_REQUEST_FAILED
Connection failed due to general or unclassified error

IP_SIP_REQUEST_NETWORK_ERROR
Connection failed due to network error or local failure

IP_SIP_REQUEST_RETRY_FAILED
Failure in request retry logic; retry not attempted

IP_SIP_REQUEST_TIMEOUT
Connection failed due to connection timeout

If retry is not enabled in a particular circumstance, or if the retry attempt failed, the Global Call library generates a GCEV_EXTENSION event containing the parameter set ID IPSET_SIP_REQUEST_ERROR, and the parameter ID IPPARM_SIP_SVC_UNAVAIL. The parameter values are the same as those listed for the "retry continuing" event.

The following code illustrates how an application can extract the failure cause information from the Extension events associated with SIP transport failures. The example assumes that the event has already been received.

```
   switch(pextensionBlk->ext_id)
   {
   .
   .
   .
      case IPSET_SIP_REQUEST_ERROR:
         ProcessRequestError(l_pParmData);
         break;
      .
      .
      .

void ProcessRequestError(GC_PARM_DATA  *parmp)
{
   REQUEST_ERROR RE;
   memcpy(&RE,parmp->value_buf,parmp->value_size);
   switch (parmp->parm_ID)
   {
      case IPPARM_SIP_DNS_CONTINUE:
         printf(" Received IPPARM_SIP_DNS_CONTINUE on %s ", RE.Method);
         break;

      case IPPARM_SIP_SVC_UNAVAIL:
         printf(" Received IPPARM_SIP_SVC_UNAVAIL on %s ",RE.Method);
         break;

      default:
         printf(" Received Unknown Request error");
         break;
   }

   switch(RE.Error)
   {
      case IP_SIP_REQUEST_NETWORK_ERROR:
         printf("IP_SIP_REQUEST_NETWORK_ERROR\n");
         break;

      case IP_SIP_REQUEST_TIMEOUT:
         printf("IP_SIP_REQUEST_TIMEOUT\n");
         break;

      case IP_SIP_REQUEST_503_RCVD:
         printf("IP_SIP_REQUEST_503_RCVD\n");
         break;

      case IP_SIP_REQUEST_FAILED:
         printf("IP_SIP_REQUEST_FAILED\n");
         break;

      default:
         printf(" Received Unknown Error cause\n");
         break;
   }
}
```

4.10 Sending and Receiving SIP INFO Messages

The SIP INFO message (as specified in IETF RFC 2976) provides a means for transporting application-level, session-related control information along the SIP signaling path after the setup of a SIP-controlled session has begun. INFO messages can be sent on an early INVITE-initiated SIP dialog (after a 101-199 provisional response) or on a confirmed dialog. The information of interest to the application can be contained in standard message header fields, proprietary header fields, or one or more MIME-encoded body parts. The Global Call library provides facilities for sending and receiving INFO requests and responses on a "pass-through" basis, meaning that there are no Global Call state changes associated with such messages. The library generates Call Info events to notify applications of incoming INFO messages, and Extension events for incoming INFO response messages. The **gc_Extension( )** Send Message API is used for outgoing INFO requests and responses.

Only one INFO request can be pending on a dialog. Once an INVITE request has been sent, another one cannot be sent until a response has been received.

The following topics discuss how applications can send, receive, and respond to INFO requests.

- Sending an INFO Message
- Receiving a Response to an INFO Message
- Receiving an INFO Message
- Responding to an INFO Message

Note: Application access to the header fields in INFO messages requires that the mask value IP_SIP_MSGINFO_ENABLE must be set into the sip_msginfo_mask field of the IP_VIRTBOARD configuration data structure before **gc_Start( )** is called. Additionally, INFO messages frequently utilize MIME message bodies, and the ability to access MIME data must be enabled by setting the IP_SIP_MIME_ENABLE mask value in the same sip_msginfo_mask.

4.10.1 Sending an INFO Message

To send an INFO message, the application begins by creating a GC_PARM_BLK that contains an element with the IPSET_MSG_SIP parameter set ID, the IPPARM_MSGTYPE parameter ID and the IP_MSGTYPE_SIP_INFO parameter value. The application adds elements for the desired header fields and one or more MIME body parts, if appropriate, to the parameter block, then uses the **gc_Extension( )** function to send the message. The header may include any combination of standard header fields and proprietary header fields. The technique for sending these header fields is described in Section 4.6.5, "Setting SIP Header Fields for Outbound Messages", on page 141. The technique for constructing MIME-encoded body parts is described in Section 4.7, "Using MIME Bodies in SIP Messages (SIP-T)", on page 146.

The following standard header fields are generally required for INFO messages:

- To (IPPARM_TO_HDR)
- From (IPPARM_FROM_HDR)
- Contact (IPPARM_CONTACT_URI, IPPARM_CONTACT_DISPLAY)
- Request-URI (IPPARM_REQUEST_URI)

- Diversion (IPPARM_DIVERSION_URI)
- Call-ID (IPPARM_CALLID_HDR)

Note: If the application does not explicitly set the Request-URI, the library populates it with the URI from the To header field by default.

The following standard header fields are also commonly used in INFO requests:

- Content-Disposition (IPPARM_CONTENT_DISPOSITION)
- Content-Encoding (IPPARM_CONTENT_ENCODING)

Note: The Content-Length and Content-Type header fields are normally filled in by the library and should not be set by the application.

The following code snippet illustrates the essential steps for constructing and sending an INFO request. The example assumes that a GC_PARM_BLK has already been declared.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_INFO);

// Insert SIP Call ID field
gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_CALLID_HDR,
                        strlen(m_CurrentCallID),
                        m_CurrentCallID);

// Insert other SIP header information here
   .
   .
   .

// transmit INFO message to network
retval = gc_Extension(GCTGT_GCLIB_CHAN, devh, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC);
   .
   .
   .
// outbound INFO has been sent.
// expect to receive a GCEV_EXTENSION containing a response
```

4.10.2 Receiving a Response to an INFO Message

After an INFO message is sent, the SIP stack will receive a response message and the library will generate a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG to notify the application.

The GC_PARM_BLK associated with Extension event wil contain a parameter element with the set ID IPSET_MSG_SIP, the parameter ID IPPARM_MSGTYPE, and one of the following two values:

- IP_MSGTYPE_SIP_INFO_OK
- IP_MSGTYPE_SIP_INFO_FAILED

The application can also retrieve the specific SIP response code from the Extension event's parameter block using the IPSET_MSG_SIP parameter set ID and the parameter ID IPPARM_MSG_SIP_RESPONSE_CODE.

Note: The application must retrieve the necessary SIP message header information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information is no longer available from the metaevent buffer.

The following code snippet illustrates the procedure for extracting the INFO response information from an Extension event.

```
//  An outbound SIP INFO request has been sent previously
//  expect an inbound SIP INFO response

switch(metaeventp->evttype)
{
   case GCEV_EXTENSION:
      while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
      {
         switch (parmp->set_ID)
         {
            // Handle SIP message information
            case IPSET_MSG_SIP:
               switch (parmp->parm_ID)
               {
                  // determine message type
                  case IPPARM_MSGTYPE:
                     MessType = (int)(*(parmp->value_buf));
                     switch (MessType)
                     {
                        case IP_MSGTYPE_SIP_INFO_OK:
                           // process INFO response
                           break;

                        case IP_MSGTYPE_SIP_INFO_FAILED:
                           // process INFO response
                           break;
                     }
                     break;

                   //  get the SIP response code
                  case IPPARM_MSG_SIP_RESPONSE_CODE:
                     ResponseCode = (int)(*(parmp->value_buf));
                     break;
               }
               break;
         }
      }
      break;
}
```

4.10.3 Receiving an INFO Message

When the SIP stack receives an incoming SIP INFO message, it generates a GCEV_CALLINFO event to the application.

The application can extract standard message header fields from the parameter block associated with the GCEV_CALLINFO event using the technique described in Section 4.6.6, "Retrieving SIP Message Header Fields", on page 144. If the message contains MIME-encoded information in its body (as many INFO messages do), the application can use the technique described in Section 4.7.3, "Getting MIME Information", on page 149 to extract the information.

Note: The application must retrieve the necessary SIP message header and body information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

The following code snippet illustrates the essential process for extracting INFO message header information from a Call Info event.

```
switch(metaeventp->evttype)
{
   case GCEV_CALLINFO:
      pParmBlock = (GC_PARM_BLK*)(metaeventp->extevtdatap);
      parmp = NULL;

      /* going thru each parameter block data*/
      while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
      {
         switch (parmp->set_ID)
         {
            /* Handle SIP message information */
            case IPSET_SIP_MSGINFO:
               switch (parmp->parm_ID)
               {
                  case IPPARM_REQUEST_URI:
                     strncpy(requestURI,(char*)parmp->value_buf,parmp->value_size);
                     sprintf(str, "gc_util_next_parm() Success, Request URI = %s",requestURI);
                     break;
                  case IPPARM_CONTACT_URI:
                     .
                     .
                     break;
                  case IPPARM_DIVERSION_URI:
                     .
                     .
                     break;
                   .
                   .
               }
               break;
            .
            .
         // etc.
            .
            .
         }
         break;
      }
      break;
}
```

4.10.4 Responding to an INFO Message

Once an application has received a GCEV_CALLINFO event for a SIP INFO message and extracted the information from the event, it must send a response message.

The response is sent as an Extension message using the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and one of the following two parameter values:

- IP_MSGTYPE_SIP_INFO_OK
- IP_MSGTYPE_SIP_FAILED

In addition, the application can set a specific SIP response code in the response message using the IPSET_MSG_SIP set ID and IPPARM_MSG_SIP_RESPONSE_CODE parameter ID:

- For an “OK” response, the value of the response code should be in the range 200 to 299; if the application does not set this parameter, the Global Call library fills in the default value 200.
- For a “Failed” response, the value of the response code should be 300 or higher; if the application does not set this parameter, the Global Call library fills in the default value 501.

The following two code snippets illustrate how an application would send “OK” and “Failed” responses to INFO messages.

“OK” Response to INFO Message

```
// inbound SIP INFO request has been received
// reply to INFO with an OK

gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_INFO_OK);

// Insert SIP response code
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        200);

// transmit INFO response message to network
retval = gc_Extension(GCTGT_GCLIB_CHAN, devh, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC);
```

“Failed” Response to INFO Message

```
// application has just received an inbound SIP INFO request.
// in this case, we are sending a "Not Implemented" failure response

gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_INFO_FAILED);

// Insert SIP response code
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        501);

// transmit INFO response message to network
retval = gc_Extension(GCTGT_GCLIB_CHAN, devh, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC);
```

4.11 Sending and Receiving SIP OPTIONS Messages

The SIP OPTIONS method provides a means for a SIP User Agent to query the capabilities of another UA or proxy, either within or outside of a SIP dialog. As an example, a client can use the OPTIONS method to discover the content types, extensions, methods, codecs, etc. that are supported by another party without having to "ring" the party by sending an INVITE.

RFC 3261 requires all user agents to support the OPTIONS method. The default behavior of the Global Call library is to send automatic responses to incoming OPTIONS requests and not provide facilities for applications to send OPTIONS requests. Optionally, an IPT virtual board can be configured to enable application access to OPTIONS messages. When access is enabled, applications can send OPTIONS requests to remote parties and are responsible for responding to incoming OPTIONS requests.

The following topics describe the Global Call library's implementation of support for the OPTIONS method.

- Default OPTIONS Behavior
- Enabling Application Access to OPTIONS Messages
- Sending OPTIONS Requests
- Receiving Responses to OPTIONS Requests
- Receiving OPTIONS Requests
- Responding to OPTIONS Requests

4.11.1 Default OPTIONS Behavior

If the SIP OPTIONS access feature is not enabled when the IPT virtual board device is started, the SIP stack in the Global Call library responds to incoming OPTIONS requests automatically, using default information, because all SIP User Agents are required to support the OPTIONS method. The application has no control over the content of these automatic response messages, nor can it send OPTIONS requests.

The default response sent by Global Call is a 200 OK, assuming that there is a channel available to handle the incoming request; if there is no channel available, Global Call responds with 486 Busy Here. The 200 OK message includes an SDP message body (Content-type: application/sdp) that indicates the same capabilities that the library reports in outgoing INVITE requests.

The default Allow header will be the following if supplementary services (call transfer) is not enabled:

```
Allow: INVITE, CANCEL, ACK, BYE
```

or the following if supplementary services is enabled:

```
Allow: INVITE, CANCEL, ACK, BYE, REFER, NOTIFY
```

Note that in either case, OPTIONS is not included in the list.

4.11.2 Enabling Application Access to OPTIONS Messages

The ability to send and respond to SIP OPTIONS requests under application control is an optional feature that can be enabled or disabled at the time that the **gc_Start( )** function is called.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** utility functions, which must be called before the **gc_Start( )** function, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The default values of two fields in the IP_VIRTBOARD structure must be overridden to enable application access to OPTIONS messages:

- The E_SIP_OPTIONS_Access field must be set to ENUM_Enabled. The default value is ENUM_Disabled, which disables access to OPTIONS messages.
- The sip_msginfo-mask field must be set to the OR of IP_SIP_MSGINFO_ENABLE and IP_SIP_MIME_ENABLE. The default mask value disables access to the header fields and MIME bodies of SIP messages, which would prevent the application from doing anything useful with OPTIONS messages.

See the reference page for IP_VIRTBOARD on page 394 for more information on these fields.

The following code snippet provides an example of enabling OPTIONS access for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
ip_virtboard[0].E_SIP_OPTIONS_Access = ENUM_Enabled;
ip_virtboard[1].E_SIP_OPTIONS_Access = ENUM_Enabled;
```

Note that in addition to enabling OPTIONS access, SIP message information access, and SIP MIME access before the virtual board is started, the application must also register the six additional SIP headers that it will need to access in OPTIONS-related messages it receives (Accept, Accept-Encoding, Accept-Language, Allow, Require, and Supported). This registration is performed on a one-time basis after the virtual board has been started, as described in Section 4.6.4, “Registering SIP Header Fields to be Retrieved”, on page 138.

4.11.3 Sending OPTIONS Requests

When SIP OPTIONS access is enabled, applications use **gc_Extension( )** to send the message after assembling the appropriate header fields and any MIME body parts in a GC_PARM_BLK. To build an OPTIONS request, the application uses the parameter set ID IPSET_MSG_SIP, the parameter ID IPPARM_MSGTYPE, and the parameter value IP_MSGTYPE_SIP_OPTIONS.

The application can send an OPTIONS message outside of a SIP dialog by using a board device handle in the **gc_Extension( )** call:

```
gc_Extension(GCTGT_GCLIB_CHAN, boarddevhandle, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

Alternatively, the application can send an OPTIONS request within a dialog by using the line device handle in the **gc_Extension( )** call:

```
gc_Extension(GCTGT_GCLIB_CHAN, linedevhandle, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

When SIP OPTIONS access is enabled, the Allow header field will be the following if supplementary services (call transfer) is not enabled:

Allow: INVITE, CANCEL, ACK, BYE, OPTIONS

or the following if supplementary services is enabled:

Allow: INVITE, CANCEL, ACK, BYE, REFER, NOTIFY, OPTIONS

The application can add additional methods to the Allow header, but the Global Call library will ensure that all of the methods supported by the library are included.

The following parameters in a GC_PARM_DATA block with a parameter set ID of IPSET_SIP_MSGINFO are used to set the header fields in the OPTIONS message, using the general techniques described in Section 4.6.5, "Setting SIP Header Fields for Outbound Messages":

parm_ID	value_buf	Default value
IPPARM_TO	To header field	Based on destination
IPPARM-REQUEST_URI	Request header URI	Derived from To header
IPPARM_FROM	From header field	Based on source
IPPARM_CONTACT_URI	Contact header URI	-none-
IPPARM_SIP_HDR	Accept header field	"Accept: application/sdp"
IPPARM_SIP_HDR	Accept-encoding header field	"Accept-encoding: " †
IPPARM_SIP_HDR	Accept-language header field	"Accept-language: en"
IPPARM_SIP_HDR	Supported header field	List of extensions supported by Global Call
IPPARM_SIP_HDR	Allow header field	List of methods supported by Global Call
IPPARM_SIP_HDR	Require header field	-none-
IPPARM_CALLID_HDR	Call-ID header field	Generated by Global Call

An empty Accept-encoding field value is permissible and equivalent to "Accept-encoding: identity", meaning no encoding

Note: The IP Call Control library automatically inserts a MIME body part containing SDP data that reflects the current capability set (that is, the same SDP information that would be sent in an INVITE request). This is the case even though the SDP information is not meaningful to the User Agent that will receive the OPTIONS request (since an OPTIONS request is not part of a negotiation).

Once the header fields are set up, the application can send the message within a call using:

```
gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

where `crn` is the CRN returned on a **gc_MakeCall( )** or in a GCEV_OFFERED event.

Or it can send the message outside a dialog using:

```
gc_Extension(GCTGT_GCLIB_CHAN, boardh, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

where `boardh` is the handle obtained by opening the board device.

The following pseudo-code shows a more complete example of constructing and sending an OPTIONS request.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_TO,
                        strlen(szTo)+1),
                        szTo);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_REQUEST_URI,
                        strlen(szRURI)+1),
                        szRURI);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_FROM,
                        strlen(szFrom)+1),
                        szFrom);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_CONTACT_URI,
                        strlen(szCntct)+1),
                        szCntct);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAccept)+1),
                        szAccept);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAcceptE)+1),
                        szAcceptE);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAcceptL)+1),
                        szAcceptL);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szSupp)+1),
                        szSupp);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAllow)+1),
                        szAllow);
```

```
gc_Extension(GCTGT_GCLIB_CHAN,
             devhandle,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

4.11.4 Receiving Responses to OPTIONS Requests

When the Global Call library's SIP stack receives a response to a SIP OPTIONS request, it generates a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The GC_PARM_BLK associated with the Extension event will contain a parameter element with the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and one of the following values:

- IP_MSGTYPE_SIP_OPTIONS_OK
- IP_MSGTYPE_SIP_OPTIONS_FAILED

The application can also retrieve the specific SIP response code from the event's parameter block using the IPSET_MSG_SIP set ID and the IPPARM_MSG_SIP_RESPONSE_CODE parameter ID.

In the case of an IP_MSGTYPE_SIP_OPTIONS_OK response, the application can use the techniques described in Section 4.6.6, "Retrieving SIP Message Header Fields" to retrieve message header fields of interest, including:

- Request-URI (IPPARM_REQUEST_URI)
- To header field (IPPARM_TO)
- From header field (IPPARM_FROM)
- Contact URI (IPPARM_CONTACT_URI)
- Accept header field (IPPARM_SIP_HDR)
- Accept-encoding header field (IPPARM_SIP_HDR)
- Accept-language header field (IPPARM_SIP_HDR)
- Supported header field (IPPARM_SIP_HDR)
- Allow header field (IPPARM_SIP_HDR)
- Require header field (IPPARM_SIP_HDR)
- Call-ID header field (IPPARM_CALLID_HDR)

The application can also extract any MIME information from the message body using the techniques described in Section 4.7.3, “Getting MIME Information”, on page 149. Note that responses to OPTIONS requests are the single case where the MIME part containing SDP information is exposed to the application rather than handled internally by the Global Call library. The SDP information is identified by the string “Content-type: application/sdp”.

In the case of an IP_MSGTYPE_SIP_OPTIONS_FAILED response, the application can use the techniques described in Section 4.6.6, “Retrieving SIP Message Header Fields” to retrieve the following message header fields:

- Request-URI (IPPARM_REQUEST_URI)
- To header field (IPPARM_TO)
- From header field (IPPARM_FROM)
- Contact URI (IPPARM_CONTACT_URI)

Note: The application must retrieve the necessary SIP message header and body information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

The following pseudo-code illustrates how to extract “OK” and “Failed” responses to OPTIONS requests from a GCEV_EXTENSION event.

```
char siphdr[IP_SIP_HDR_MAXLEN];
char AcceptHeader[IP_SIP_HDR_MAXLEN];
char Accept_encodingHeader[IP_SIP_HDR_MAXLEN];
char Accept_languageHeader[IP_SIP_HDR_MAXLEN];


case   GCEV_EXTENSION:
   if( pextensionBlk->ext_id== IPEXTID_RECEIVEMSG )
   {
      while ((l_pParm  = gc_util_next_parm(pParmBlock, l_pParm )) != 0)
      {
         int l_mtype=  (int)(*( l_pParm ->value_buf));
         switch (l_pParm ->set_ID)
         {
            case IPSET_MSG_SIP:
               if(l_pParm ->parm_ID == IPPARM_MSGTYPE)
               {
                  if(l_mtype== IP_MSGTYPE_SIP_OPTIONS_OK)
                  {
                     printf("OPTIONS request successful\n");
                  }
                  else if (l_mtype== IP_MSGTYPE_SIP_ OPTIONS_FAILED)
                  {
                     printf("OPTIONS request failedl\n");
                  }
               }
               else if(l_pParm ->parm_ID == PARM_MSG_SIP_RESPONSE_CODE)
               {
                  int *l_RC= (int *) l_pParm ->value_buf;
                  printf ("Response Code %d \n",*l_RC);
               }
            case IPSET_SIP_MSGINFO:
               switch(l_pParm ->parm_ID)
               {
                  case IPPARM_SIP_HDR:
                     strncpy(siphdr,(char*)parmp->value_buf,parmp->value_size);
                     siphdr[parmp->value_size]='\0';
                     if(!strnicmp(siphdr,"Accept-encoding",strlen("Accept-encoding" )))
                     {
                        strcpy(Accept_encodingHeader,siphdr);
                     }
                     else if (! strnicmp(siphdr,"Accept-language",strlen("Accept-language")))
                     {
                        strcpy(Accept_languageHeader,siphdr);
                     }
                     else if (! strnicmp(siphdr,"Accept",strlen("Accept")))
                     {
                        strcpy(AcceptHeader,siphdr);
                     }
                  …
                  //(process other headers)
            default :
               break;
         }
```

4.11.5 Receiving OPTIONS Requests

When the Global Call library’s SIP stack receives a SIP OPTIONS request, it generates an Extension event (GCEV_EXTENSION) of type IPEXTID_RECEIVEMSG. The GC_PARM_BLK associated with the Extension event will contain a parameter element with the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and the value IP_MSGTYPE_SIP_OPTIONS.

The application can use the techniques described in Section 4.6.6, “Retrieving SIP Message Header Fields” to retrieve header fields of interest, including:

- To header field (IPPARM_TO)
- Request URI (IPPARM_REQUEST_URI)
- From header field (IPPARM_FROM)
- Contact URI (IPPARM_CONTACT_URI)
- Accept header field (IPPARM_SIP_HDR)
- Accept-encoding header field (IPPARM_SIP_HDR)
- Accept-language header field (IPPARM_SIP_HDR)
- Supported header field (IPPARM_SIP_HDR)
- Allow header field (IPPARM_SIP_HDR)
- Require header field (IPPARM_SIP_HDR)
- Call-ID header field (IPPARM_CALLID_HDR)

The application can also extract MIME information from the message body using the techniques described in Section 4.7.3, “Getting MIME Information”, on page 149. Note that the MIME part that contains SDP information is **not** exposed to the application.

Note: The application must retrieve the necessary SIP message header and body information by copying the data into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

The following pseudo-code illustrates how to extract an OPTIONS request from a received GCEV_EXTENSION event,

```
case   GCEV_EXTENSION:
   if( pextensionBlk->ext_id== IPEXTID_RECEIVEMSG)
   {
      while ((l_pParm = gc_util_next_parm(pParmBlock, l_pParm )) != 0)
      {
         int l_mtype=  (int)(*( l_pParm->value_buf));
         switch (l_pParm->set_ID)
         {
            case IPSET_MSG_SIP:
               if(l_pParm ->parm_ID == IPPARM_MSGTYPE)
               {
                  if(l_mtype== IP_MSGTYPE_SIP_OPTIONS )
                  {
                     printf("OPTIONS request received\n");
                  }
               …
               }
               break
            case IPSET_SIP_MSGINFO:
               switch(l_pParm ->parm_ID)
               {
                  case  IPPARM_CALLID_HDR:
                     strncpy(g_CurrentCallID,(char*)parmp->value_buf,parmp->value_size);
                     g_CurrentCallID[parmp->value_size]='\0';
                     break;
                  …
                  //(process other headers)
               default :
         break;
      }
   }
```

4.11.6 Responding to OPTIONS Requests

If SIP OPTIONS access is enabled, it is the application’s responsibility to respond to incoming OPTIONS requests, assuming that there is a channel available to handle the incoming request. (If there is no channel available, Global Call automatically responds with 486 Busy Here.)

OPTIONS responses are sent as Global Call Extension messages using **gc_Extension( )**. There are separate message types for “OK and “Failed” response messages, but both types **must** use the Call-ID header obtained from the received request.

“Success” Response Message

“OK” responses to OPTIONS requests use the IPSET_MSG_SIP / IPPARM_MSGTYPE parameter set and ID with a value of IP_MSGTYPE_SIP_OPTIONS_OK.

The following parameters in the parameter set IPSET_SIP_MSGINFO are used to set the header fields in the OPTIONS response message, using the general techniques described in Section 4.6.5, “Setting SIP Header Fields for Outbound Messages”:

parm_ID	value_buf	Default value
IPPARM_CONTACT_URI	Contact header URI	-none-
IPPARM_SIP_HDR	Accept header field	“application/sdp”
IPPARM_SIP_HDR	Accept-encoding header field	“ “
IPPARM_SIP_HDR	Accept-language header field	“en”

parm_ID	value_buf	Default value
IPPARM_SIP_HDR	Supported header field	List of extensions supported by Global Call
IPPARM_SIP_HDR	Allow header field	List of methods supported by Global Call
IPPARM_SIP_HDR	Require header field	-none-
IPPARM_CALLID_HDR	Call-ID header field	Generated by Global Call

The Global Call library ensures that the Allow header field contains all SIP methods supported by the library, which includes the following methods if supplementary services (call transfer) is not enabled:

INVITE, CANCEL, ACK, BYE, OPTIONS

or the following if supplementary services is enabled:

INVITE, CANCEL, ACK, BYE, REFER, NOTIFY, OPTIONS

When sending an "OK" response, the IP Call Control library automatically inserts a MIME body part that contains SDP data which reflects the current capability set (that is, the same SDP information that would be sent in an INVITE request). This may be the standard capability set, or the application may explicitly configure the capabilities to send in the "OK" by using the GCSET_CHAN_CAPABILITY set ID and the IPPARM_LOCAL_CAPABILITY parameter ID:

```
gc_util_insert_parm_ref(&target_datap,
                        GCSET_CHAN_CAPABILITY,
                        IPPARM_LOCAL_CAPABILITY,
                        sizeof(IP_CAPABILITY),
                        &a_DefaultCapability);
```

The application can also send generic, non-SDP MIME information using the techniques described in Section 4.7.4, "Sending MIME Information", on page 154.

The following pseudo-code illustrates the general procedure for constructing a successful response to an OPTIONS request.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS_OK);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAccept)+1),
                        szAccept);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_CALLID_HDR,
                        strlen(g_CurrentCallID)+1,
                        g_CurrentCallID);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAcceptE)+1),
                        szAcceptE);
```

```
gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAcceptL)+1),
                        szAcceptL);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szSupp)+1),
                        szSupp);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAllow)+1),
                        szAllow);

//insert a message body

gc_Extension(GCTGT_GCLIB_CHAN,
             devhandle,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

“Failed” Response Message

“Failed” responses to OPTIONS requests use the IPSET_MSG_SIP set ID and IPPARM_MSGTYPE parameter ID with a value of IP_MSGTYPE_SIP_OPTIONS_FAILED.

When sending the response message, the application **must** include the Call-ID header field value that was retrieved from the incoming OPTIONS request. The response is on the board device (that is, the **gc_Extension( )** call uses the board handle that was obtained when opening the board device), and the Call-ID is used to identify the specific request to which the response applies.

The application can also set a specific SIP response code in a “Failed” OPTIONS response message using IPSET_MSG_SIP / IPPARM_MSG_SIP_RESPONSE_CODE. If the application does not set a specific response code, Global Call uses the default value 486 (Busy Here).

The following pseudo-code illustrates sending a “Failed” response with the response code 486.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS_FAILED);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_CALLID_HDR,
                        strlen(g_CurrentCallID)+1,
                        g_CurrentCallID);

gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        486);
```

```
gc_Extension(GCTGT_GCLIB_CHAN,
             boardh,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

The following pseudo-code illustrates sending a “Failed” response with the response code 415, which requires Accept, Accept-Encoding, and Accept-Language header fields.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS_FAILED);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAccept)+1),
                        szAccept);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_CALLID_HDR,
                        strlen(g_CurrentCallID)+1,
                        g_CurrentCallID);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAcceptE)+1),
                        szAcceptE);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        strlen(szAcceptL)+1),
                        szAcceptL);

gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        415);

gc_Extension(GCTGT_GCLIB_CHAN,
             boardh,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

4.12 Using SIP SUBSCRIBE and NOTIFY Messages

The SIP SUBSCRIBE and NOTIFY methods (as documented in IETF RFC 3265) provide a basic mechanism for event notification between nodes. In the most basic implementation, an entity on a network can use the SUBSCRIBE request to communicate its interest in certain state changes for resources or calls on the network, and those entities (or other entities acting on their behalf) can send NOTIFY messages as notifications when those state changes occur. This SUBSCRIBE / NOTIFY mechanism is used outside of a dialog or call.

In addition, there may be unsubscribed NOTIFY messages that are not preceded by a corresponding SUBSCRIBE request. One common use of unsubscribed NOTIFY messages is to enable and disable the Message Waiting Indicator (MWI) on a PIMG.

The Global Call call control library for SIP fully supports both the SUBSCRIBE and NOTIFY methods, including both subscribed and unsubscribed NOTIFY. These messages are all handled on a "pass-through" basis (in other words, there are no Global Call state changes associated with the events). The Global Call Extension API mechanism is used in all cases. Outgoing requests and responses are sent by building an appropriate GC_PARM_BLK and then calling **gc_Extension( )**, while incoming requests and responses are passed to the application as GCEV_EXTENSION events.

Note that the NOTIFY messages which are used in the Global Call library implementation of SIP call transfer are not handled explicitly by applications using the techniques described in this section. The Global Call library handles these messages implicitly, automatically generating the outgoing NOTIFY messages that are required in a call transfer operation, and passing incoming NOTIFY messages associated with a call transfer to the application as GCEV_INVOKE_XFER or GCEV_INVOKE_XFER_FAIL events. The exception to this generalization is a NOTIFY message that is sent to the Transferor after the primary call has been dropped; in this case, the message is interpreted as a "normal" NOTIFY outside of a dialog and is passed as a GCEV_EXTENSION event that the application must explicitly accept or reject as described in Section 4.12.8, "Responding to NOTIFY Requests", on page 191. These post-termination NOTIFY messages may occur under various circumstances, including the following:

- In the normal course of events in the scenario where the Transferor is notified upon ringing of the transferred call (see Figure 22, "Successful SIP Unattended Call Transfer, Party A Notified with Ringing", on page 73)
- If a 200 OK to NOTIFY is lost in the network and the primary call is terminated by party A before party B sends another NOTIFY as a retry
- If a non-Global Call UA sends a NOTIFY for some reason after the primary call is terminated

Note that an application that will be sending and receiving SUBSCRIBE and NOTIFY messages must enable both the SIP message information (header) and SIP MIME (body) access features before starting the IPT virtual board with the **gc_Start( )** function. The **INIT_IP_VIRTBOARD( )** utility function populates the IP_VIRTBOARD structure with default values. The default values of the sip_msginfo_mask field in this structure must be overridden to enable application access to SUBSCRIBE and NOTIFY messages. Specifically, the sip_msginfo_mask field must be set to the OR of IP_SIP_MSGINFO_ENABLE and IP_SIP_MIME_ENABLE. See the reference page for IP_VIRTBOARD on page 394 for more information on this field and these mask values.

The following code snippet provides an example of enabling message header and body access for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
```

The following topics describe how applications send, receive, and respond to SUBSCRIBE and NOTIFY requests:

- Sending SUBSCRIBE Requests
- Receiving Responses to SUBSCRIBE Requests
- Receiving SUBSCRIBE Requests
- Responding to SUBSCRIBE Requests
- Sending NOTIFY Requests
- Receiving Responses to NOTIFY Requests
- Receiving NOTIFY Requests
- Responding to NOTIFY Requests

4.12.1 Sending SUBSCRIBE Requests

To send a SUBSCRIBE request message, the application begins by creating a GC_PARM_BLK that contains an element with the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID and the IP_MSGTYPE_SIP_SUBSCRIBE parameter value. The application adds elements for the desired header fields and one or more MIME body parts, if appropriate, to the parameter block, then uses the **gc_Extension( )** function to send the message. The header may include any combination of standard header fields and proprietary header fields. General techniques for setting header fields are described in Section 4.6.5, "Setting SIP Header Fields for Outbound Messages". The technique for constructing MIME body parts is described in Section 4.7.4, "Sending MIME Information".

The header fields that normally must be set in a SUBSCRIBE request include the following:

- To display string (IPPARM_TO_DISPLAY)
- From display string (IPPARM_FROM_DISPLAY)
- Expires header field (IPPARM_EXPIRES_HDR)
- Event header field (IPPARM_EVENT_HDR)
- Call-ID header field (IPPARM_CALLID_HDR)

SUBSCRIBE requests normally contain an Expires header field, which indicates the duration of the subscription. When the application does not explicitly set an Expires header field, the default duration that is defined in the SIP "event package" for the particular type of event will apply. To keep a subscription effective beyond the accepted duration, the subscriber needs to send a new SUBSCRIBE message on the same dialog when it receives an expiration message. To terminate or unsubscribe an existing subscription, the application can send a SUBSCRIBE request with the value 0 in the Expires header field to specify immediate expiration.

The following code snippet illustrates how an application constructs and sends a SUBSCRIBE request.

```
void CSubNotMgr::SendSIPSubscribe (char* pRequestURI,
                                   char* pTo,
                                   char* pFrom,
                                   char* pExpire,
                                   char* pEvent,
                                   char* pCallID)

{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP SUBSCRIBE\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP  parmblkp = NULL;   // input parameter block pointer
   GC_PARM_BLKP  retblkp = NULL;    // return parameter block
   GC_INFO       gc_error_info;     // GlobalCall error information data
   int           retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_SUBSCRIBE);

   // Insert SIP request URI field
   if (pRequestURI)
   {
      gc_util_insert_parm_ref(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_REQUEST_URI,
                              strlen(pRequestURI),
                              pRequestURI);
   }

   // Insert SIP To field
   if (pTo)
   {
      gc_util_insert_parm_ref(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_TO_DISPLAY,
                              strlen(pTo),
                              pTo);
   }

   // Insert SIP From field
   if (pFrom)
   {
      gc_util_insert_parm_ref(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_FROM_DISPLAY,
                              strlen(pFrom),
                              pFrom);
   }

   // Insert SIP Expire field
   if (pExpire)
   {
      gc_util_insert_parm_ref(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_EXPIRES_HDR,
                              strlen(pExpire),
                              pExpire);
   }
```

```
      // Insert SIP Event field
      if (pEvent)
      {
         gc_util_insert_parm_ref(&parmblkp,
                                 IPSET_SIP_MSGINFO,
                                 IPPARM_EVENT_HDR,
                                 strlen(pEvent),
                                 pEvent);
      }

      // Insert SIP Call ID field
      if (pCallID)
      {
         gc_util_insert_parm_ref(&parmblkp,
                                 IPSET_SIP_MSGINFO,
                                 IPPARM_CALLID_HDR,
                                 strlen(pCallID),
                                 pCallID);
      }

      if (parmblkp == NULL)
      {
         // memory allocation error
         return;
      }

      // transmit SUBSCRIBE message to network
      retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                            IPEXTID_SENDMSG, parmblkp,
                            &retblkp, EV_ASYNC);

      if (retval != GC_SUCCESS)
      {
         gc_ErrorInfo( &gc_error_info );
         printf ("Error : gc_Extension() on HANDLE: 0x%lx,
                 GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
                 CC ErrorValue: 0x%lx - %s\n", boardh,
                 gc_error_info.gcValue, gc_error_info.gcMsg,
                 gc_error_info.ccLibId, gc_error_info.ccLibName,
                 gc_error_info.ccValue, gc_error_info.ccMsg);
         return;
      }

      // clean up
      gc_util_delete_parm_blk(parmblkp);

      m_bSubscribeSent = true;
   }
```

4.12.2 Receiving Responses to SUBSCRIBE Requests

After a SUBSCRIBE request is sent, the remote entity responds with an accept or reject reply, which the call control library passes to the application as a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The value of the IPSET_MSG_SIP / IPPARM_MSGTYPE parameter in the Extension event for a SUBSCRIBE response is one of the following two values:

- IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT
- IP_MSGTYPE_SIP_SUBSCRIBE_REJECT

Additionally, the subscriber application may periodically receive an event that indicates the expiration of the subscription duration. Note that the application does not have to respond to an expiration message because the message indicates that the transaction is no longer active. For such an expiration message, the value of the IPSET_MSG_SIP / IPPARM_MSGTYPE parameter in the Extension event is:

- IP_MSGTYPE_SIP_SUBSCRIBE_EXPIRE

Note: The application must retrieve the necessary SIP message header information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information is no longer available from the metaevent buffer.

The following example code illustrates the general procedure for extracting information from the Extension event for any of the incoming messages associated with the SUBSCRIBE and NOTIFY methods.

```
// main event loop
// get a GCEV_EXTENSION event and process it
void process_event(void)
{
   METAEVENT  metaevent;
   int        evttype;

   gc_GetMetaEvent(&metaevent);
   evttype = metaevent.evttype;

   GC_PARM_BLK  *pParmBlock = NULL;
   GC_PARM_DATA *parmp = NULL;

   switch (evttype)
   {
      case GCEV_EXTENSION:
         OnExtensionEvent(&metaevent);
         break;
   }
}

// process GCEV_EXTENSION event
// get SIP Msg and SIP Msg Info
void OnExtensionEvent(METAEVENT *metaeventp)
{
   GC_PARM_BLK      *pParmBlock = NULL;
   EXTENSIONEVTBLK  *pExtensionBlock = NULL;
   GC_PARM_DATA     *parmp = NULL;

   pExtensionBlock = (EXTENSIONEVTBLK*)(metaeventp->extevtdatap);
   pParmBlock = &pExtensionBlock->parmblk;

   parmp = NULL;
   int CurrentMessage = 0;

   // going thru each parameter block data
   while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
   {
      switch (parmp->set_ID)
      {
         // Handle SIP message information
         case IPSET_MSG_SIP:
            CurrentMessage = ProcessSIPMsg(parmp);
            break;
         /* Handle SIP message information */
         case IPSET_SIP_MSGINFO:
            ProcessSIPMsgInfo(parmp);
            break;

         default:
            break;
      }
   }

   pParmBlock = (GC_PARM_BLK*)(metaeventp->extevtdatap);
   parmp = NULL;
}

// determine type of SIP Message and process accordingly
int CSubNotMgr::ProcessSIPMsg(GC_PARM_DATA  *parmp)
{
   int MessType=0;
   switch (parmp->parm_ID)
   {
      case IPPARM_MSGTYPE:
      {
         MessType = (int)(*(parmp->value_buf));
         switch (MessType)
         {
            case IP_MSGTYPE_SIP_SUBSCRIBE:
               // process here
               break;
            case IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT:
               // process here
               break;
            case IP_MSGTYPE_SIP_SUBSCRIBE_REJECT:
               // process here
               break;
            case IP_MSGTYPE_SIP_SUBSCRIBE_EXPIRE:
               // process here
               break;
            case IP_MSGTYPE_SIP_NOTIFY:
               // process here
               break;
            case IP_MSGTYPE_SIP_NOTIFY_ACCEPT:
               // process here
               break;
            case IP_MSGTYPE_SIP_NOTIFY_REJECT:
               // process here
               break;
            default:
               break;
         }
         break;
      }
      default:
         break;
   }
   return MessType;
}

// process SIP Msg Info
void CSubNotMgr::ProcessSIPMsgInfo(GC_PARM_DATA  *parmp)
{
   char   requestURI[IP_REQUEST_URI_MAXLEN];
   char   contactURI[IP_CONTACT_URI_MAXLEN];
   char   diversionURI[IP_DIVERSION_URI_MAXLEN];
   char   event[IP_EVENT_HDR_MAXLEN];
   char   expires[IP_EXPIRES_HDR_MAXLEN];
   switch (parmp->parm_ID)
   {
      case IPPARM_REQUEST_URI:
         strncpy(requestURI,(char*)parmp->value_buf,parmp->value_size);
         requestURI[parmp->value_size]='\0';
         break;
      case IPPARM_CONTACT_URI:
         strncpy(contactURI,(char*)parmp->value_buf,parmp->value_size);
         contactURI[parmp->value_size]='\0';
         break;
      case IPPARM_DIVERSION_URI:
         strncpy(diversionURI,(char*)parmp->value_buf,parmp->value_size);
         diversionURI[parmp->value_size]='\0';
         break;
      case IPPARM_EVENT_HDR:
         strncpy(event,(char*)parmp->value_buf,parmp->value_size);
         event[parmp->value_size]='\0';
         break;
      case IPPARM_EXPIRES_HDR:
         strncpy(expires,(char*)parmp->value_buf,parmp->value_size);
         expires[parmp->value_size]='\0';
         break;
      case IPPARM_CALLID_HDR:
         strncpy(m_CurrentCallID,(char*)parmp->value_buf,parmp->value_size);
         m_CurrentCallID[parmp->value_size]='\0';
         break;
      default:
         break;
   }
}
```

4.12.3 Receiving SUBSCRIBE Requests

When the SIP stack receives a SIP SUBSCRIBE request, the Global Call library generates an Extension event of type IPEXTID_RECEIVEMSG. The value of the IPSET_MSG_SIP / IPPARM_MSGTYPE parameter in the Extension event is IP_MSGTYPE_SIP_SUBSCRIBE in this case.

The application can use the techniques described in Section 4.6.6, “Retrieving SIP Message Header Fields” to retrieve message header fields of interest, including:

- To display string (IPPARM_TO_DISPLAY)
- From display string (IPPARM_FROM_DISPLAY)
- Expires header field (IPPARM_EXPIRES_HDR)
- Event header field (IPPARM_EVENT_HDR)
- Call-ID header field (IPPARM_CCALLID_HDR)

If the message has a body, the application can extract the MIME-encoded information using the techniques described in Section 4.7.3, “Getting MIME Information”.

Note: The application must retrieve the necessary SIP message header and body information by copying the data into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

A code example that illustrates the general procedure for retrieving information from all incoming messages associated with the SUBSCRIBE and NOTIFY methods is included in Section 4.12.2, "Receiving Responses to SUBSCRIBE Requests", on page 181.

4.12.4 Responding to SUBSCRIBE Requests

Once an application has received a GCEV_EXTENSION event for a SIP SUBSCRIBE request and extracted the information from the event, it must send a response message.

The response is sent as an Extension message using the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and one of the following two parameter values:

- IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT
- IP_MSGTYPE_SIP_SUBSCRIBE_REJECT

The "Accept" message is a 200 OK, while the "Reject" message is a 501 response. In either case, the response message **must** include the Call-ID header field value that was received in the SUBSCRIBE request so that the subscriber can match the response to the request.

The following two code snippets illustrate how an application would send "Accept" and "Reject" responses to SUBSCRIBE requests.

"Accept" response to SUBSCRIBE request

When accepting a SUBSCRIBE request, a SIP entity normally includes an Expires header field, which may contain the same value that was received in the Expires header field of the SUBSCRIBE request or any smaller value.

```
void CSubNotMgr::SendSIPSubscribeAccept (char* pExpire)
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP SUBSCRIBE Accept\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP   parmblkp = NULL;   // input parameter block pointer
   GC_PARM_BLKP   retblkp = NULL;    // return parameter block
   GC_INFO        gc_error_info;     // GlobalCall error information data
   int            retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT);

   // Insert SIP Expire field
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_EXPIRES_HDR,
                           strlen(pExpire),
                           pExpire);
   // Insert SIP Call ID field
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           strlen(m_CurrentCallID),
                           m_CurrentCallID);

   if (parmblkp == NULL)
   {
      // memory allocation error
      return;
   }

   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bSubscribeAcceptSent = true;
}
```

“Reject” response to SUBSCRIBE request

```
void CSubNotMgr::SendSIPSubscribeReject (void)
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP SUBSCRIBE Reject\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP   parmblkp = NULL;   // input parameter block pointer
   GC_PARM_BLKP   retblkp = NULL;    // return parameter block
   GC_INFO        gc_error_info;     // GlobalCall error information data
   int            retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_SUBSCRIBE_REJECT);

   // Insert SIP Call ID field
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           strlen(m_CurrentCallID),
                           m_CurrentCallID);
   if (parmblkp == NULL)
   {
      // memory allocation error
      return;
   }
   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bSubscribeRejectSent = true;
}
```

4.12.5 Sending NOTIFY Requests

To send a NOTIFY message, the application begins by creating a GC_PARM_BLK that contains an element with the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and the IP_MSGTYPE_SIP_NOTIFY parameter value. The application adds elements for the desired header fields and one or more MIME body parts, if appropriate, to the parameter block, then uses the **gc_Extension( )** function to send the message. The header fields that can be set and the general technique for setting them are described in Section 4.6.5, “Setting SIP Header Fields for Outbound Messages”. The technique for constructing MIME bodies is described in Section 4.7.4, “Sending MIME Information”.

The header fields that normally must be set in a NOTIFY request include the following:

- To display string (IPPARM_TO_DISPLAY)
- From display string (IPPARM_FROM_DISPLAY)
- Event header field (IPPARM_EVENT_HDR)
- Call-ID header field (IPPARM_CCALLID_HDR)

If the NOTIFY being sent is a subscribed NOTIFY, the Call-ID header field must contain the same Call-ID value as the SUBSCRIBE request that the NOTIFY relates to.

The following code snippet illustrates how an application constructs and sends a NOTIFY request.

```
void CSubNotMgr::SendSIPNotify ( char* pRequestURI,
                                 char* pTo,
                                 char* pFrom,
                                 char* pEvent,
                                 char* pBody,
                                 char* pCallID)

{
   char str[MAX_STRING_SIZE];
   char *pBlankBody = " ";
   sprintf(str, "<--- Sending SIP NOTIFY on device %d\n", hsendboard);
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);
```

```
GC_PARM_BLKP parmblkp = NULL;    // input parameter block pointer
GC_PARM_BLKP parmblkbody = NULL; // body parms
GC_PARM_BLKP retblkp = NULL;     // return parameter block
GC_INFO      gc_error_info;      // GlobalCall error information data
int          retval = GC_SUCCESS;

// Insert SIP message type
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_NOTIFY);

// Insert SIP Request-URI
if (pRequestURI)
{
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_REQUEST_URI,
                           strlen(pRequestURI),
                           pRequestURI);
}

// Insert SIP To field
if (pTo)
{
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_TO_DISPLAY,
                           strlen(pTo),
                           pTo);
}

// Insert SIP From field
if (pFrom)
{
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_FROM_DISPLAY,
                           strlen(pFrom),
                           pFrom);

   //Insert SIP Contact header field
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CONTACT_URI,
                           strlen(pFrom),
                           pFrom);
}

// Insert SIP Event field
if (pEvent)
{
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_EVENT_HDR,
                           strlen(pEvent),
                           pEvent);
}
```

```
// Insert SIP CallID field
if (pCallID)
{
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           strlen(pCallID),
                           pCallID);
}

// Insert the message Body
if (pBody)
{

   // Insert Content-Type field
   // Add 1 to strlen for the NULL termination character
   gc_util_insert_parm_ref(&parmblkbody,
                           IPSET_MIME,
                           IPPARM_MIME_PART_TYPE,
                           (unsigned char)(strlen(pBody) + 1),
                            pBody);

   // Insert Body Size
   gc_util_insert_parm_val(&parmblkbody,
                           IPSET_MIME,
                           IPPARM_MIME_PART_BODY_SIZE,
                           sizeof(unsigned long),
                           strlen(pBlankBody));

   // Insert MIME part Body Pointer
   gc_util_insert_parm_val(&parmblkbody,
                           IPSET_MIME,
                           IPPARM_MIME_PART_BODY,
                           sizeof(unsigned long),
                           (unsigned long)pBlankBody);

   // Insert parm block B pointer to parm block A
   gc_util_insert_parm_val(&parmblkp, //pParmBlockA,
                           IPSET_MIME,
                           IPPARM_MIME_PART,
                           sizeof(unsigned long),
                           (unsigned long)parmblkbody);

   if (parmblkbody == NULL)
   {
      // memory allocation error
      return;
   }
}

if (parmblkp == NULL)
{
   // memory allocation error
   return;
}

// transmit NOTIFY message to network
retval = gc_Extension(GCTGT_GCLIB_CHAN,
                      hsendboard,
                      IPEXTID_SENDMSG,
                      parmblkp,
                      &retblkp,
                      EV_ASYNC);
   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s,
              CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n",
              boardh,
              gc_error_info.gcValue,
              gc_error_info.gcMsg,
              gc_error_info.ccLibId,
              gc_error_info.ccLibName,
              gc_error_info.ccValue,
              gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);
   if (pBody) gc_util_delete_parm_blk(parmblkbody);

   m_bNotifySent=true;
}
```

4.12.6 Receiving Responses to NOTIFY Requests

After a NOTIFY request is sent, the remote entity responds with an accept or reject reply, which the call control library sends to the application as a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The value of the IPSET_MSG_SIP / IPPARM_MSGTYPE parameter in the GC_PARM_BLK associated with the Extension event for a NOTIFY response is one of the following two values:

- IP_MSGTYPE_SIP_NOTIFY_ACCEPT
- IP_MSGTYPE_SIP_NOTIFY_REJECT

Note: The application must retrieve the necessary SIP message header information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information is no longer available from the metaevent buffer.

A code example that illustrates the general technique for retrieving information from all incoming messages associated with the SUBSCRIBE and NOTIFY methods is included in Section 4.12.2, “Receiving Responses to SUBSCRIBE Requests”, on page 181.

4.12.7 Receiving NOTIFY Requests

When the SIP stack receives a SIP NOTIFY request, the Global Call library generates an Extension event (GCEV_EXTENSION) of type IPEXTID_RECEIVEMSG. The IPSET_MSG_SIP / IPPARM_MSGTYPE parameter in the Extension event in this case has the value IP_MSGTYPE_SIP_NOTIFY. Both subscribed and unsubscribed NOTIFY requests can be received; in the case of a subscribed NOTIFY, the value of the Call-ID header field will match the Call-ID of a previously sent SUBSCRIBE request.

The application can use the techniques described in Section 4.6.6, “Retrieving SIP Message Header Fields” to retrieve message header fields of interest, including:

- To display string (IPPARM_TO_DISPLAY)
- From display string (IPPARM_FROM_DISPLAY)
- Event header field (IPPARM_EVENT_HDR)
- Call-ID header field (IPPARM_CCALLID_HDR)

If the message has a body, the application can extract the MIME-encoded information using the techniques described in Section 4.7.3, “Getting MIME Information”.

Note: The application must retrieve the necessary SIP message header and body information by copying the data into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

A code example that illustrates the general procedure for retrieving information from all incoming messages associated with the SUBSCRIBE and NOTIFY methods is included in Section 4.12.2, “Receiving Responses to SUBSCRIBE Requests”, on page 181.

4.12.8 Responding to NOTIFY Requests

Once an application has received a GCEV_EXTENSION event for a SIP NOTIFY message (either subscribed or unsubscribed) and extracted the information from the event, it must send a response message.

The response is sent as an Extension message using the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and one of the following two parameter values:

- IP_MSGTYPE_SIP_NOTIFY_ACCEPT
- IP_MSGTYPE_SIP_NOTIFY_REJECT

For an “Accept” response the message sent is a 200 OK, while “Reject” sends a 501 response. In either case, the response message must include the Call-ID header that was received in the NOTIFY request.

The following two code snippets illustrate how an application would send “Accept” and “Reject” responses to NOTIFY requests.

“Accept” Response to NOTIFY Request

```
void CSubNotMgr::SendSIPNotifyAccept ()
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP NOTIFY Accept\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP   parmblkp = NULL;  // input parameter block pointer
   GC_PARM_BLKP   retblkp = NULL;   // return parameter block
   GC_INFO        gc_error_info;    // GlobalCall error information data
   int            retval = GC_SUCCESS;
   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_NOTIFY_ACCEPT);

   // Insert SIP Call ID field
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           strlen(m_CurrentCallID),
                           m_CurrentCallID);

   if (parmblkp == NULL)
   {
      // memory allocation error
      return;
   }

   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bNotifyAcceptSent = true;
}
```

“Reject” Response to NOTIFY Request

```
void CSubNotMgr::SendSIPNotifyReject (void)
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP NOTIFY Reject\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP  parmblkp = NULL; // input parameter block pointer
   GC_PARM_BLKP  retblkp = NULL;  // return parameter block
   GC_INFO       gc_error_info;   // GlobalCall error information data
   int           retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_NOTIFY_REJECT);

   // Insert SIP Call ID field
   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           strlen(m_CurrentCallID),
                           m_CurrentCallID);
   if (parmblkp == NULL)
   {
      // memory allocation error
   return;
   }

   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bNotifyRejectSent = true;
}
```

4.13 Handling DTMF

DTMF handling is described under the following topics:

- Specifying DTMF Support
- Getting Notification of DTMF Detection
- Generating DTMF
- Generating or Detecting DTMF Tones Using a Voice Resource

4.13.1 Specifying DTMF Support

Global Call can be used to configure which DTMF transmission modes are supported by the application. The DTMF mode can be specified in one of three ways:

- for all line devices simultaneously by using **gc_SetConfigData( )**
- on a per-line device basis by using **gc_SetUserInfo( )** with a **duration** parameter value of GC_ALLCALLS
- on a per-call basis by using **gc_SetUserInfo( )** with a **duration** parameter value of GC_SINGLECALL

The GC_PARM_BLK associated with the **gc_SetConfigData( )** or **gc_SetUserInfo( )** function is used to indicate which DTMF modes are supported. The GC_PARM_BLK should include the IPSET_DTMF set ID, the IPPARM_SUPPORT_DTMF_BITMASK parameter ID, and a parameter value that specifies DTMF transmission mode(s). The parameter value may be a single bitmask value or the OR of more than one value to specify multiple supported modes.

Note: The IPPARM_SUPPORT_DTMF_BITMASK parameter can only be replaced rather than modified. For each **gc_SetConfigData( )** or **gc_SetUserInfo( )** call, the previous value of the IPPARM_SUPPORT_DTMF_BITMASK parameter is overwritten.

Bitmask values for SIP

SIP applications **must** set the DTMF signaling mode before calling **gc_MakeCall( )**, **gc_AnswerCall( )**, **gc_AcceptCall( )**, or **gc_CallAck( )**. If a SIP application does not do this, the function call fails with an IPERR_NO_DTMF_CAPABILITY indication. Supported bitmask values are:

IP_DTMF_TYPE_INBAND_RTP
DTMF digits are sent and received inband via standard RTP transcoding.

IP_DTMF_TYPE_RFC_2833
DTMF digits are sent and received in the RTP stream as defined in RFC 2833.

Bitmask values for H.323

An H.323 application that supports only the default H.245 User Input Indication (UII) Alphanumeric mode does not need to explicitly set the DTMF signaling mode. All other applications must set the DTMF mode using the following bitmask values:

IP_DTMF_TYPE_ALPHANUMERIC (default)
DTMF digits are sent and received in H.245 UII Alphanumeric messages.

Note: HMP only supports the H.245 UII Alphanumeric mode; H.245 UII Signal mode is **not** supported.

IP_DTMF_TYPE_INBAND_RTP
DTMF digits are sent and received inband via standard RTP transcoding.

IP_DTMF_TYPE_RFC_2833
DTMF digits are sent and received in the RTP stream as defined in RFC 2833.

As an example, the following code snippet shows how to specify the out-of-band signaling mode for all calls on a line device:

```
{
   GC_PARM_BLKP parmblkp = NULL;
   gc_util_insert_parm_val(&parmblkp,
                           IPSET_DTMF,
                           IPPARM_SUPPORT_DTMF_BITMASK,
                           sizeof(char),
                           IP_DTMF_TYPE_INBAND_RTP);

   if (gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[callindex].ldev,
                      parmblkp, GC_ALLCALLS) != GC_SUCCESS) {

          // gc_SetUserInfo returned an error
}
gc_util_delete_parm_blk(parmblkp);
```

The mode in which DTMF is transmitted (Tx) is determined by the intersection of the mode values specified by the IPPARM_SUPPORT_DTMF_BITMASK and the receive capabilities of the remote endpoint. When this intersection includes multiple modes, the selected mode is based on the following priority:

1. RFC 2833
2. H.245 UII Alphanumeric (H.323 only)
3. Inband

The mode in which DTMF is received (Rx) is based on the selection of transmission mode from the remote endpoint; however, RFC 2833 can only be received if RFC 2833 is specified by the IPPARM_SUPPORT_DTMF_BITMASK parameter ID.

Table 14 summarizes the DTMF mode settings and associated behavior.

Table 14. Summary of DTMF Mode Settings and Behavior

IP_DTMF_TYPE_ RFC_2833	IP_DTMF_TYPE_ ALPHANUMERIC†	IP_DTMF_TYPE_ INBAND	Transmit (Tx) DTMF Mode	Receive (Rx) DTMF Mode
1 (enabled)	0 (disabled)	0 (disabled)	RFC 2833 if supported by remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
0 (disabled)	1 (enabled)	0 (disabled)	UII Alphanumeric†	UII Alphanumeric† or Inband as chosen by the remote endpoint
0 (disabled)	0 (disabled)	1 (enabled)	Inband	UII Alphanumeric† or Inband as chosen by the remote endpoint
0 (disabled)	1 (enabled)	1 (enabled)	UII Alphanumeric†	UII Alphanumeric† or Inband as chosen by the remote endpoint
1 (enabled)	1 (enabled)	0 (disabled)	RFC 2833 if supported by remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
† Applies to H.323 only.				

Table 14. Summary of DTMF Mode Settings and Behavior (Continued)

IP_DTMF_TYPE_ RFC_2833	IP_DTMF_TYPE_ ALPHANUMERIC†	IP_DTMF_TYPE_ INBAND	Transmit (Tx) DTMF Mode	Receive (Rx) DTMF Mode
1 (enabled)	0 (disabled)	0 (disabled)	RFC 2833 if supported by remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
1 (enabled)	0 (disabled)	1 (enabled)	RFC 2833 if supported by the remote endpoint, otherwise Inband	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
1 (enabled)	1 (enabled)	1 (enabled)	RFC 2833 if supported by the remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
† Applies to H.323 only.				

When using RFC 2833, the payload type is specified using the IPSET_DTMF set ID, the IPPARM_DTMF_RFC2833_PAYLOAD_TYP parameter ID, and one of the following values:

- IP_USE_STANDARD_PAYLOADTYPE (default payload type, 101)
- Any value in the range 96 to 127 (dynamic payload type)

4.13.2 Getting Notification of DTMF Detection

Once DTMF support has been configured (see Section 4.13.1, “Specifying DTMF Support”, on page 193), the application can specify which DTMF modes will provide notification when DTMF digits are detected. The events for this notification must be enabled; see Section 4.17, “Enabling and Disabling Unsolicited Notification Events”, on page 205.

Once the events are enabled, when an incoming DTMF digit is detected, the application receives a GCEV_EXTENSION event, with an extID of IPEXTID_RECEIVE_DTMF. The GCEV_EXTENSION event contains the digit and the method. The GC_PARM_BLK associated with the event contains the IPSET_DTMF parameter set ID and the following parameter ID:

IPPARM_DTMF_ALPHANUMERIC

For H.323, DTMF digits are received in H.245 User Input Indication (UII) alphanumeric messages. The parameter value is of type IP_DTMF_DIGITS. See the reference page for IP_DTMF_DIGITS on page 390 for more information. For SIP, this parameter is **not** supported.

4.13.3 Generating DTMF

Once DTMF support has been configured (see Section 4.13.1, “Specifying DTMF Support”, on page 193), the application can use the **gc_Extension( )** function to generate DTMF digits. The relevant **gc_Extension( )** function parameter values in this context are:

- **target_type** should be GCTGT_GCLIB_CRN
- **target_id** should be the actual CRN
- **ext_ID** should be IPEXTID_SEND_DTMF

The GC_PARM_BLK pointed to by the **parmblkp** parameter must contain the IPSET_DTMF parameter set ID and the following parameter ID:

IPPARM_DTMF_ALPHANUMERIC
: For H.323, specifies that DTMF digits are to be sent in H.245 User Input Indication (UII) Alphanumeric messages. For SIP, this parameter is **not** supported.

4.13.4 Generating or Detecting DTMF Tones Using a Voice Resource

Using a voice resource to generate or detect DTMF tones in Inband or RFC2833 DTMF transfer mode requires that the voice resource (for example, dxxxB1C1) be attached to the IPT network device (for example, iptB1T1) that also has an IP Media device (ipmB1C1) attached. This can be achieved using the **gc_OpenEx( )** function as follows:

```
gc_OpenEx(lindevice, ":P_IP:N_iptB1T1:M_ipmB1C1:V_dxxxB1C1", EV_ASYNC, userattr)
```

where:

- linedevice is a Global Call device
- P_IP indicates that the device supports both the H.323 and SIP protocols
- N_iptB1T1 identifies the IPT network device
- M_ipmB1C1 identifies the IPT Media device
- V_dxxxB1C1 specifies the voice resource that will be used to generate or detect the DTMF tones
- EV_ASYNC indicates the function operates in asynchronous mode
- userattr points to a buffer where user information can be stored

Note: Alternatively, the IPT network device and IP Media device can be opened without the voice resource, and the IP line device can be routed to the voice device when needed.

Once the voice resource is attached to the IPT network and IPT Media devices, the following voice library functions can be used:

- **dx_dial( )** to generate DTMF tones
- **dx_getdig( )** to detect DTMF tones

4.14 Getting Media Streaming Status and Connection Information

The application can receive notification of changes in the status (connection and disconnection) of media streaming in the transmit and receive directions as GC_EXTENSIONEVT events. When the event is a notification of the connection of the media stream in either direction, information about the coders negotiated for that direction and the local and remote RTP addresses is also available.

The events for this notification must be enabled by setting or adding the bitmask value EXTENSIONEVT_SIGNALING_STATUS to the GC_EXTENSIONEVT mask; see Section 4.17, “Enabling and Disabling Unsolicited Notification Events”, on page 205. Once the events are enabled, when a media streaming connection state changes, the application receives a GCEV_EXTENSION event. The EXTENSIONEVTBLK structure pointed to by the extevtdatap pointer within the GCEV_EXTENSION event will contain the following information for all media streaming status changes:

extID
    IPEXTID_MEDIAINFO

parmblk
    A GC_PARM_BLK containing the protocol connection status with the IPSET_MEDIA_STATE parameter set ID and one of the following parameter IDs:

- IPPARM_TX_CONNECTED – Media streaming has been initiated in transmit direction. The value for this parameter ID is datatype IP_CAPABILITY and contains the coder configuration that resulted from the capability exchange with the remote peer.
- IPPARM_TX_DISCONNECTED – Media streaming has been terminated in transmit direction. No parameter value is used with this parameter ID.
- IPPARM_RX_CONNECTED – Media streaming has been initiated in receive direction. The value for this parameter ID is datatype IP_CAPABILITY and contains the coder configuration that resulted from the capability exchange with the remote peer.
- IPPARM_RX_DISCONNECTED – Media streaming has been terminated in receive direction. No parameter value is used with this parameter ID.

When the parameter value in the GC_PARM_BLK structure is IPPARM_TX_CONNECTED, indicating that a transmit media stream connection has occurred, the GC_PARM_BLK structure will also contain the local and remote RTP addresses. These addresses are handled as an RTP_ADDR data structure, which contains both the port number and the IP address. The parameter set ID used for the RTP addresses is IPSET_RTP_ADDRESS, and the parameter IDs are IPPARM_LOCAL and IPPARM_REMOTE.

RTP Address and Coder Information Retrieval Example

The following code snippet illustrates how to retrieve the RTP addresses and negotiated coder information from a media stream connection event:

```
//When the event is an extension event:

GC_PARM_BLKP     gcParmBlk;
EXTENSIONEVTBLK  *pextensionBlk;
GC_PARM_DATA     *parmp = NULL;
RTP_ADDR         l_RTA1,l_RTA2;
pextensionBlk = (EXTENSIONEVTBLK *)(m_pMetaEvent->extevtdatap);
gcParmBlk = (&(pextensionBlk->parmblk));

GC_PARM_DATAP l_pParmData;
IP_CAPABILITYl_IPCap;

switch(pextensionBlk->ext_id)
{
   case IPEXTID_MEDIAINFO:

   //get the coder info:
   l_pParmData = gc_util_find_parm(gcParmBlk, IPSET_MEDIA_STATE, IPPARM_TX_CONNECTED);

   if(l_pParmData != NULL)
   {
      memcpy(&l_IPCap, l_pParmData->value_buf, l_pParmData->value_size);

      // get the local RTP address:
      l_pParmData= gc_util_find_parm(gcParmBlk, IPSET_RTP_ADDRESS, IPPARM_LOCAL);
      if(l_pParmData!= NULL)
      {
         memcpy(&l_RTA1,l_pParmData->value_buf,l_pParmData->value_size);
      }

      //get the remote RTP address:
      l_pParmData =gc_util_find_parm(gcParmBlk, IPSET_RTP_ADDRESS, IPPARM_REMOTE);
      if(l_pParmData != NULL)
      {
         memcpy(&l_RTA2, l_pParmData->value_buf, l_pParmData->value_size);
      }
   }

   else
   {
      //only get tx or rx, not both
      l_pParmData = gc_util_find_parm(gcParmBlk, IPSET_MEDIA_STATE, IPPARM_RX_CONNECTED);
      if(l_pParmData != NULL)
      {
         memcpy(&l_IPCap, l_pParmData->value_buf, l_pParmData->value_size);
      }
   }
}
```

4.15 Getting Notification of Underlying Protocol State Changes

The application can receive notification of intermediate protocol signaling state changes for both H.323 and SIP. The events for this notification must be enabled; see Section 4.17, “Enabling and Disabling Unsolicited Notification Events”, on page 205.

Once these events are enabled, when a protocol state change occurs, the application receives a GCEV_EXTENSION event. The EXTENSIONEVTBLK structure pointed to by the extevtdatap pointer within the GCEV_EXTENSION event will contain the following information:

extID
IPEXTID_IPPROTOCOL_STATE

parmblk
A GC_PARM_BLK containing the protocol connection status with the IPSET_IPPROTOCOL_STATE parameter set ID and one of the following parameter IDs:

- IPPARM_SIGNALING_CONNECTED – The signaling for the call has been established with the remote endpoint. For example, in H.323, a CONNECT message was received by the caller or a CONNECTACK message was received by the callee.
- IPPARM_SIGNALING_DISCONNECTED – The signaling for the call has been terminated with the remote endpoint. For example, in H.323, a RELEASE message was received by the terminator or a RELEASECOMPLETE message was received by peer.
- IPPARM_CONTROL_CONNECTED – Media control signaling for the call has been established with the remote endpoint. For example, in H.323, an OpenLogicalChannel message (for the receive direction) or an OpenLogicalCahnnelAck message (for the transmit direction) was received.
- IPPARM_CONTROL_DISCONNECTED – Media control signaling for the call has been terminated with the remote endpoint. For example, in H.323, an EndSession message was received.

Note: The parameter value field in this GC_PARM_BLK in each case is unused (NULL).

4.16 Sending Protocol Messages

The following message types are supported:

- Nonstandard User Input Indication (UII) Message (H.245)
- Nonstandard Facility Messages (Q.931)
- Nonstandard Registration Messages

Table 15 summarizes the set IDs and parameter IDs used to send the messages and describes the call states in which each message should be sent.

Table 15. Summary of Protocol Messages that Can be Sent

Type	Set ID & Parameter ID	When Message Should be Sent
Nonstandard UII Message (H.245)	IPSET_MSG_H245 • IPPARM_MSGTYPE (set to IP_MSGTYPE_H245_INDICATION)	Only when call is in Connected state
Nonstandard Facility Message (Q.931)	IPSET_MSG_Q931 • IPPARM_MSGTYPE (set to IP_MSGTYPE_Q931_FACILITY)	In any call state
Nonstandard Registration Message	IPSET_MSG_RAS • IPPARM_MSGTYPE (set to IP_MSGTYPE_REG_NONSTD	

4.16.1 Nonstandard UII Message (H.245)

To send nonstandard UII messages, use the **gc_Extension( )** function in asynchronous mode with an **ext_id** (extension ID) of IPEXTID_SENDMSG. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. At the sending end, reception of a GCEV_EXTENSIONCMPLT event indicates that the message has been sent. At the receiving end, a GCEV_EXTENSION event with the same ext_id value is generated. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSION event is a pointer to an EXTENSIONEVTBLK structure which in turn contains a GC_PARM_BLK that includes all of the data in the message.

The relevant set IDs and parameter IDs for this purpose are:

IPSET_MSG_H245
IPPARM_MSGTYPE
- value = IP_MSGTYPE_H245_INDICATION

IPSET_NONSTANDARDDATA
with either:
IPPARM_NONSTANDARDDATA_DATA
- value = actual nonstandard data.
  Maximum length = MAX_NS_PARM_DATA_LENGTH (128)

IPPARM_NONSTANDARDDATA_OBJID
- value = object ID string.
  Maximum length = MAX_NS_PARM_OBJID_LENGTH (40)

or:
IPPARM_NONSTANDARDDATA_DATA
- value = actual nonstandard data.
  Maximum length = MAX_NS_PARM_DATA_LENGTH (128)

IPPARM_H221NONSTANDARD
- value = H.221 nonstandard data identifier

Note: The message type (IPPARM_MSGTYPE) is mandatory. At least one other information element must be included.

See Section 8.2.14, “IPSET_MSG_Q931”, on page 366 and Section 8.2.18, “IPSET_NONSTANDARDDATA”, on page 368 for more information.

```
.
.
.
/* H245 UII with ObjId and data */

rc = gc_util_insert_parm_val(&t_PrmBlkp, IPSET_MSG_H245, IPPARM_MSGTYPE,
                             sizeof(int), IP_MSGTYPE_H245_INDICATION);

rc = gc_util_insert_parm_ref(&t_PrmBlkp, IPSET_NONSTANDARDDATA,
                             IPPARM_NONSTANDARDDATA_OBJID, ObjLen+1, ObjId);

rc = gc_util_insert_parm_ref(&t_PrmBlkp, IPSET_NONSTANDARDDATA,
                             IPPARM_NONSTANDARDDATA_DATA, DataLen+1, data);
```

```
if (rc == -1)
{
   printf("Fail to insert parm");
   return -1;
}
else
   printf("Sending IP H245 UII Message");

gc_Extension(GCTGT_GCLIB_CRN,
             crn,
             IPEXTID_SENDMSG,
             t_PrmBlkp,
             &t_RetBlkp,
             EV_ASYNC);

gc_util_delete_parm(t_PrmBlkp);
.
.
.
```

4.16.2 Nonstandard Facility Message (Q.931)

Use the **gc_Extension( )** function in asynchronous mode with an **ext_id** (extension ID) of IPEXTID_SENDMSG to send nonstandard facility (Q.931 Facility) messages. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. At the sending end, a GCEV_EXTENSIONCMPLT event is received indicating that the message has been sent. At the receiving end, a GCEV_EXTENSION event with the same ext_id value is generated. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSION event is a pointer to an EXTENSIONEVTBLK structure which in turn contains a GC_PARM_BLK that includes all of the data in the message.

The relevant parameter set IDs and parameter IDs are:

IPSET_MSG_Q931
  IPPARM_MSGTYPE
    Set to IP_MSGTYPE_Q931_FACILITY

IPSET_NONSTANDARDDATA
  with either:
  IPPARM_NONSTANDARDDATA_DATA
  - value = Actual nonstandard data
    Maximum length = MAX_NS_PARM_DATA_LENGTH (128).

  IPPARM_NONSTANDARDDATA_OBJID
  - value = object ID string.
    Maximum length = MAX_NS_PARM_OBJID_LENGTH (40).

  or:
  IPPARM_NONSTANDARDDATA_DATA
  - value = actual nonstandard data.
    Maximum length = MAX_NS_PARM_DATA_LENGTH (128).

  IPPARM_H221NONSTANDARD
  - value = H.221 nonstandard data identifier

Note: The message type (IPPARM_MSGTYPE) is mandatory. At least one other information element must be included.

See Section 8.2.14, “IPSET_MSG_Q931”, on page 366 and Section 8.2.18, “IPSET_NONSTANDARDDATA”, on page 368 for more information.

The following code shows how to set up and send a Q.931 nonstandard facility message.

```
char ObjId[]= "1 22 333 4444";
char NSData[]= "DataField_Facility";

GC_PARM_BLKP    gcParmBlk = NULL;

gc_util_insert_parm_val(&gcParmBlk,
                         IPSET_MSG_Q931,
                         IPPARM_MSGTYPE,
                         sizeof(int),
                         IP_MSGTYPE_Q931_FACILITY);

gc_util_insert_parm_ref(&gcParmBlk,
                         IPSET_NONSTANDARDDATA,
                         IPPARM_NONSTANDARDDATA_OBJID,
                         sizeof(ObjId),
                         ObjId);

gc_util_insert_parm_ref(&gcParmBlk,
                         IPSET_NONSTANDARDDATA,
                         IPPARM_NONSTANDARDDATA_DATA,
                         sizeof(NSData),
                         NSData);

gc_Extension( GCTGT_GCLIB_CRN,
              crn,
              IPEXTID_SENDMSG,
              gcParmBlk,
              NULL,
              EV_ASYNC);

gc_util_delete_parm_blk(gcParmBlk);
```

4.16.3 Nonstandard Registration Message

Use the **gc_Extension( )** function in asynchronous mode with an **ext_id** (extension ID) of IPEXTID_SENDMSG to send nonstandard registration messages. The **target_type** should be GCTGT_CCLIB_NETIF and the **target_id** should be the board device handle, since the message destination is the Gatekeeper. At the sending end, a GCEV_EXTENSIONCMPLT event is received indicating that the message has been sent. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSION event is a pointer to an EXTENSIONEVTBLK structure, which in turn contains a GC_PARM_BLK that includes all of the data in the message.

The relevant set IDs and parameter IDs for this purpose are:

IPSET_PROTOCOL
- IPPARM_PROTOCOL_BITMASK
  - value = IP_PROTOCOL_H323

IPSET_MSG_REGISTRATION
- IPPARM_MSGTYPE
  - value = IP_MSGTYPE_REG_NONSTD

IPSET_NONSTANDARDDATA
with either:
IPPARM_NONSTANDARDDATA_DATA

- value = actual nonstandard data
  Maximum length = MAX_NS_PARM_DATA_LENGTH (128)

IPPARM_NONSTANDARDDATA_OBJID

- value = object ID string
  Maximum length = MAX_NS_PARM_OBJID_LENGTH (40)

or:
IPPARM_NONSTANDARDDATA_DATA

- value = actual nonstandard data
  Maximum length = MAX_NS_PARM_DATA_LENGTH (128)

IPPARM_H221NONSTANDARD

- value = H.221 nonstandard data identifier

Note: The protocol and message type (IPPARM_MSGTYPE) are mandatory, and at least one other information element must be included.

The following code snippet illustrates how to send an H.221 nonstandard registration message.

```
{
   GC_PARM_BLKP parmblkp = NULL;
   char h221nonstd_id[] = "My H.221 Nonstandard data identifier";
                                      /* must be <= MAX_NS_PARM_OBJID_LENGTH (40) */
   char nonstd_data[] = "My nonstandard_data";

   gc_util_insert_parm_val(&parmblkp, IPSET_PROTOCOL, IPPARM_PROTOCOL_BITMASK,
                           sizeof(char), IP_PROTOCOL_H323);
   gc_util_insert_parm_val(&parmblkp, IPSET_MSG_REGISTRATION, IPPARM_MSGTYPE,
                           sizeof(unsigned long), IP_MSGTYPE_REG_NONSTD);
   gc_util_insert_parm_ref(&parmblkp, IPSET_NONSTANDARDDATA, IPPARM_NONSTANDARDDATA_DATA,
                           sizeof(nonstd_data), nonstd_data);
   gc_util_insert_parm_ref(&parmblkp, IPSET_NONSTANDARDDATA, IPPARM_H221NONSTANDARD,
                           sizeof(h221nonstd_id), h221nonstd_id);

   if (gc_Extension(GCTGT_CCLIB_NETIF, bdev, IPEXTID_SENDMSG, parmblkp, NULL,
                    EV_ASYNC) != GC_SUCCESS)
   {
      printandlog(ALL_DEVICES, GC_APIERR, NULL, "gc_Extension() Failed", 0);
       exitdemo(1);
   }
}
```

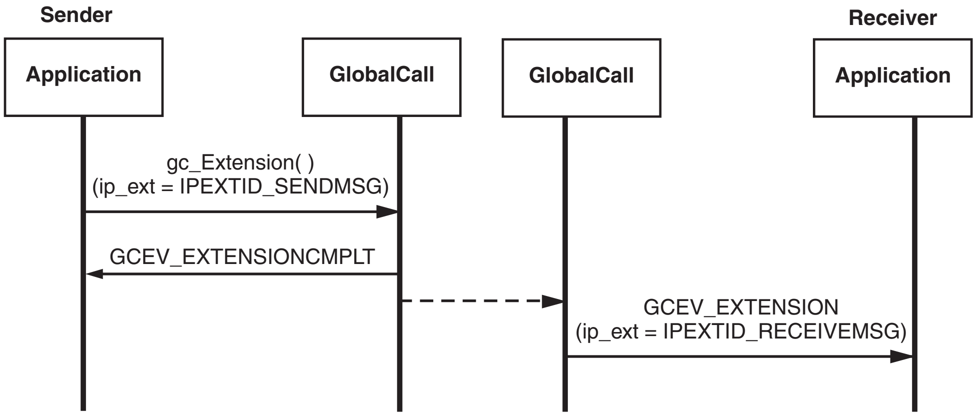
See Section 8.2.15, “IPSET_MSG_REGISTRATION”, on page 366 and Section 8.2.18, “IPSET_NONSTANDARDDATA”, on page 368 for more information.

4.16.4 Sending Facility, UII, or Registration Message Scenario

The **gc_Extension( )** function can be used to send H.245 UII messages or Q.931 nonstandard facility messages. Figure 43 shows this scenario.

An H.245 UII message can only be sent when a call is in the connected state. A Q.931 nonstandard facility message can be sent in any call state.

Figure 43. Sending Protocol Messages



4.17 Enabling and Disabling Unsolicited Notification Events

The application can enable and disable the GCEV_EXTENSION events associated with unsolicited notification including:

- DTMF digit detection
- underlying protocol (Q.931 and H.245) connection state changes
- media streaming connection state changes
- T.38 fax events

Enabling and disabling unsolicited GCEV_EXTENSION notification events is done by manipulating the event mask, which has a default value of zero, using the **gc_SetConfigData( )** function. The relevant **gc_SetConfigData( )** function parameter values in this context are:

- **target_type** – GCTGT_CCLIB_NETIF
- **target_id** – IPT board device
- **size** – set to a value of GC_VALUE_LONG
- **target_datap** – a pointer to a GC_PARM_BLK structure that contains the parameters to be configured

The GC_PARM_BLK should contain the IPSET_EXTENSIONEVT_MSK parameter set ID and one of the following parameter IDs:

GCACT_ADDMSK
Add an event to the mask

GCACT_SUBMSK
Remove an event from the mask

GCACT_SETMSK
Set the mask to a specific value

Possible values (corresponding to events that can be added or removed from the mask are) are:

EXTENSIONEVT_DTMF_ALPHANUMERIC
For notification of DTMF digits received in User Input Indication (UII) messages with alphanumeric data. When using SIP, this value is not applicable.

EXTENSIONEVT_SIGNALING_STATUS
For notification of intermediate protocol state changes in signaling (in H.323, for example, Q.931 Connected and Disconnected) and control (in H.323, for example, H.245 Connected and Disconnected).

EXTENSIONEVT_STREAMING_STATUS
For notification of the status and configuration information of transmit or receive directions of media streaming including: Tx Connected, Tx Disconnected, Rx Connected, and Rx Disconnected.

EXTENSIONEVT_T38_STATUS
For notification of fax tones detected on T.38 fax.

4.18 Configuring the Sending of the Proceeding Message

The application can configure if the Proceeding message is sent under application control (using the **gc_CallAck( )** function) or automatically by the stack. The **gc_SetConfigData( )** function can be used for this purpose.

The relevant set ID and parameter ID that must be included in the associated GC_PARM_BLK are:

GCSET_CHAN_CONFIG
GCPARM_CALLPROC. Possible values are:

- GCCONTROL_APP – The application must use **gc_CallAck( )** to send the Proceeding message. This is the default.
- GCCONTROL_TCCL – The stack sends the Proceeding message automatically.

4.19 Enabling and Disabling H.245 Tunneling

Tunneling is the encapsulation of H.245 media control messages within Q.931/H.225 signaling messages. If tunneling is enabled, one less TCP port is required for incoming connections.

For outgoing calls, the application can enable or disable tunneling by including the following parameter element in the GCLIB_MAKECALL_BLK used by the **gc_MakeCall( )** function:

IPSET_CALLINFO
IPPARM_H245TUNNELING
Possible values:

- IP_H245TUNNELING_ON
- IP_H245TUNNELING_OFF

For incoming calls, tunneling is enabled by default, but it can be configured on a board device level (where a board device is a virtual entity that corresponds to a NIC or NIC address; see Section 2.3.2, "IPT Board Devices", on page 43). This is done using the **gc_SetConfigData( )** function with target ID of the board device and the parameters above specified in the GC_PARM_BLKP structure associated with the **gc_SetConfigData( )** function.

Note: Tunneling for inbound calls can be configured on a board device basis only (using the **gc_SetConfigData( )** function). Tunneling for inbound calls **cannot** be configured on a per line device or per call basis (using the **gc_SetUserInfo( )** function).

4.20 Using H.323 Annex M Tunneled Signaling Messages

The Global Call IP call control library supports the tunneled signaling message capability that is documented in Annex M of the ITU-T recommendations for H.323. This capability allows DSS/QSIG/ISUP messages to be encapsulated in common H.225 call signaling messages. Note that this tunneled message capability is separate and distinct from H.245 tunnelling.

The tunneled signaling message capabilities are described in the following topics:

- Tunneled Signaling Message Overview
- Sending Tunneled Signaling Messages
- Enabling Reception of Tunneled Signaling Messages
- Receiving Tunneled Signaling Messages

4.20.1 Tunneled Signaling Message Overview

The ITU-T Annex M recommendation specifies that tunneled signaling message fields may be contained in a number of different H.225 messages, including SETUP, INFORMATION, CALL PROCEEDING, ALERTING, PROGRESS, NOTIFY, CONNECT, RELEASE COMPLETE, and FACILITY.

The Global Call implementation of tunneled signaling messages allows applications to send tunneled messages only in H.225 SETUP messages, as sent by the **gc_MakeCall( )** function. Only one tunneled signaling message can be sent per SETUP message.

The reception of tunneled signaling messages via Global Call is an optional feature that can only be enabled when starting the virtual board. When the feature is enabled, tunneled message fields can be retrieved from any of the H.225 messages specified in Annex M. An application has no ability to specify which message types it wishes to receive tunneled signaling message in; if there is any possibility that the remote agent could be a non-Global Call application, the local application must be prepared to handle tunneled messages in any of the specified H.225 message types .

Tunneled signaling messages are constructed by configuring a GC_PARM_BLK with parameter elements that contain protocol identification, message content, and non-standard data fields. The protocol identification can use either a protocol object ID or an alternate identification data structure, IP_TUNNELPROTOCOL_ALTID. As in other Global Call implementations of non-standard data, the H.221 protocol can be specified, or the non-standard data can be identified via an object ID. When the GC_PARM_BLK is configured, it is passed to the **gc_MakeCall( )** function as part of the GC_MAKECALL_BLK data structure, at which point the library and H.323 stack package the supplied data as tunneled signaling message fields in the H.225 SETUP message sent by the function call.

Note: The **gc_SetUserInfo( )** and **gc_SetConfigData( )** functions **cannot** be used to configure the tunneled signaling message parameters, and the **gc_Extension( )** function cannot be used to send a message that contains tunneled signaling message fields. The configured parameter data must be passed directly to **gc_MakeCall( )**.

When reception of tunneled signaling messages is enabled as described in Section 4.20.3, "Enabling Reception of Tunneled Signaling Messages", on page 210, applications must register to receive the messages using the **gc_Extension( )** function. When any H.225 message containing a tunneled signaling message is received, the library generates an asynchronous GCEV_EXTENSIONCMPLT completion event, which includes the tunneled signaling message information in the metaevent data. Tunneled signaling messages can only be retrieved within a call (the application must use a valid CRN when registering to receive tunneled signaling messages), but the call can be in any state.

4.20.2 Sending Tunneled Signaling Messages

The process of sending a tunneled signaling message begins by composing a GC_PARM_BLK that contains parameter elements for the message protocol, the message content, and any non-standard data.

The first parameter element identifies the message protocol:

IPSET_TUNNELEDSIGNALMSG
with either:
IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJID
- value = protocol object ID string

or
IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID
- value = alternate protocol ID information in an IP_TUNNELPROTOCOL_ALTID data structure

The second parameter element contains the actual message content:

IPSET_TUNNELEDSIGNALMSG
IPPARM_TUNNELEDSIGNALMSG_CONTENT
- value = actual message content

If the tunneled signal message includes non-standard data, the GC_PARM_BLOCK needs to contain two additional parameter elements. These parameters should not be inserted in the GC_PARM_BLK if non-standard data is not being sent in the message.

IPSET_TUNNELEDSIGNALMSG
IPPARM_TUNNELEDSIGNALMSG_NSDATA_DATA
- value = actual non-standard data

IPSET_TUNNELEDSIGNALMSG

with either:

IPPARM_TUNNELEDSIGNALMSG_NSDATA_OBJID

- value = non-standard data object ID string

or

IPPARM_TUNNELEDSIGNALMSG_NSDATA_H221NS

- value = H.221 non-standard data information in an IP_H221NONSTANDARD data structure

Once the GC_PARM_BLK is composed, the block is included in a GC_MAKECALL_BLK, and that block is then passed as a parameter in a call to **gc_MakeCall( )**.

The following code example illustrates the process of composing the parameter block for a tunneled signaling message.

```
#include <stdio.h>
#include <string.h>
#include <gcip.h>
#include <.h>

void main()
{
   IP_TUNNELPROTOCOL_ALTID tsmTpAltId;
   IP_H221NONSTANDARD      tsmH221NS;
   GC_PARM_BLKP            pParmBlock;

   /*. .   Main Processing….*/

   char *pTP_Oid = "itu-t (0) recommendation (0) q (17) 763";
                 // Note that the Object Id strings must be in the correct ASN.1 format.
   char *pMsgContent = "00 11 22 33 44 55";

   char TP_AltID_Type[]    = "Tunneled Protocol Alternate ID protocol type";
   char TP_AltID_Variant[] = "Tunneled Protocol Alternate ID protocol variant";
   char TP_AltID_SubId[]   = “Tunneled Protocol Alternate ID subidentifier – User";

   char *ptsmNSData_Data   = "Tunneled Signaling Message Non Standard Data";
   char *ptsmNSData_Oid    = "itu-t (0) recommendation (0) q (17) 931";
               // Note that the Object Id strings must be in the correct ASN.1 format
               // otherwise it may cause problems in the RV Stack.

   /* Initialize the structures before use */
   INIT_IP_TUNNELPROTOCOL_ALTID (&tsmTpAltId);

   strcpy(tsmTpAltId.protocolType, TP_AltID_Type);
   tsmTpAltId.protocolTypeLength = strlen(TP_AltID_Type);
   strcpy(tsmTpAltId.protocolVariant, TP_AltID_Variant);
   tsmTpAltId.protocolVariantLength = strlen(TP_AltID_Variant);
   strcpy(tsmTpAltId.subIdentifier, TP_AltID_SubId);
   tsmTpAltId.subIdentifierLength = strlen(TP_AltID_SubId);

   tsmH221NS.country_code = 91;
   tsmH221NS.extension = 202;
   tsmH221NS.manufacturer_code = 11;

   choiceOfTSMProtocol = 1;
       /* App decides whether to use the tunneled signaling message Protocol Object ID/ AltID */
   choiceOfNSData = 1;
       /* App decides which type of object identifier to use for TSM NS Data */

   /* setting tunneled signaling message fields */
```

```
if (choiceOfTSMProtocol)
/* App decides the choice of the tunneled signaling msg protocol object identifier */
/* It cannot set both objid & alternate id */
{
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_ TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJID,
                           (unsigned char) (strlen(pTP_Oid) + 1),
                           pTP_Oid);
}

else
{
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_ TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID,
                           (unsigned char)sizeof(IP_TUNNELPROTOCOL_ALTID),
                           & tsmTpAltId);
}

gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_TUNNELEDSIGNALMSG,
                        IPPARM_TUNNELEDSIGNALMSG_CONTENT,
                        (unsigned char)(strlen(pMsgContent)+1),
                        pMsgContent);



/* Now fill in the Tunneled Signaling message Non Standard data fields */
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_TUNNELEDSIGNALMSG,
                        IPPARM_TUNNELEDSIGNALMSG_NSDATA_DATA,
                        (unsigned char)(strlen(ptsmNSData_Data)+1),
                        ptsmNSData_Data);

if (choiceOfNSData)
/* App decides the CHOICE of Non Standard OBJECTIDENTIFIER. */
/* It cannot set both objid & H221 */
{
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_ TUNNELEDSIGNALMSG_NSDATA_OBJID,
                           (unsigned char) (strlen(ptsmNSData_Oid)+1),
                           ptsmNSData_Oid);
}

else
{
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_ TUNNELEDSIGNALMSG_NSDATA_H221NS,
                           (unsigned char)sizeof(IP_H221NONSTANDARD),
                           & tsmH221NS);
}

/*. .. Continue Main processing. … call gc_MakeCall() */
```

4.20.3 Enabling Reception of Tunneled Signaling Messages

The ability to retrieve tunneled signaling messages from inbound H.225 messages is an optional feature that can be enabled or disabled at the time the **gc_Start**( ) function is called.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions, which must be called before **gc_Start( )**, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The default value of the h323_msginfo_mask field in the IP_VIRTBOARD structure does not enable either access to Q.931 message information elements or the ability to receive tunneled signaling messages. To enable either or both of these features for an IPT device, the default value of the h323_msginfo_mask field must be overridden with a value that represents the appropriate logical combination of the two defined mask values. To enable reception of tunneled signaling messages, the value IP_H323_ANNEXMMSG_ENABLE must be set. The following code snippet enables Q.931 message IE access on two virtual boards and enables tunneled signaling messages on the second board only:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE;
                                    /* override Q.931 message default */
ip_virtboard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE |  IP_H323_ANNEXMMSG_ENABLE;
                                    /* override Q.931 message and TSM defaults */
```

Note: Once the tunneled signaling message feature is enabled on a virtual board, there is no way to disable the reception of the messages other than reconfiguring and restarting the virtual board.

4.20.4 Receiving Tunneled Signaling Messages

Assuming that reception of tunneled signaling messages was enabled when the virtual board was started, the application registers to receive within each call using the **gc_Extension( )** function and the extension ID IPEXTID_GETINFO.

The parameters for the **gc_Extension( )** function call must be set up as follows:

- **target_type** must be GCTGT_GCLIB_CRN. The function cannot be called for a line device.
- **target_id** must be a valid CRN. The call can be in any state.
- **ext_id** must be IPEXTID_GETINFO
- **parmblkp** must point to a GC_PARM_BLK that contains a parameter with the set ID IPSET_TUNNELEDSIGNALMSG and IPPARM_TUNNELEDSIGNALMSG_CONTENT parameter ID. This is the only field that will always be present in every received tunneled signaling message; the library automatically ensures that all tunneled signaling message fields that actually exist in the message are retrieved as long as this one parameter is present in the GC_PARM_BLK.
- **retblkp** must be a valid pointer to a GC_PARM_BLK
- **mode** must be EV_ASYNC

The following code illustrates a typical registration process.

```
int getTSMInfo(CRN crn)
{
   GC_PARM_BLKP gcParmBlk = NULL;
   GC_PARM_BLKP retParmBlk;
   int frc;
   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_TUNNELEDSIGNALMSG,
                                 IPPARM_TUNNELEDSIGNALMSG_CONTENT,
                                 sizeof(int),1);

   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_Extension (GCTGT_GCLIB_CRN,
                       crn,
                       IPEXTID_GETINFO,
                       gcParmBlk,
                       &retParmBlk,
                       EV_ASYNC);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   gc_util_delete_parm_blk(gcParmBlk);

   return GC_SUCCESS;

}
```

After this registration, when an H.225 message containing a tunneled signaling message is received by the library, it generates an asynchronous GCEV_EXTENSIONCMPLT completion event. The extevtdatap field in the METAEVENT structure for this event is a pointer to an EXTENSIONEVTBLK structure, which in turn contains a GC_PARM_BLK that contains the fields of the received tunneled signaling message. Applications are then able to extract the data of interest using code similar to the following example.

Notes: ***1.*** The application must take care to retrieve the Annex M Message information from any incoming H.225 message before the next H.225 message arrives. If the new message also contains TSM information, that new TSM overwrite the prior information.

2. Because parameter values that are contained in a GC_PARM_BLK are limited to 255 bytes in length, any received TSM data that exceeds 255 bytes will be truncated.

```
int OnExtension(GC_PARM_BLKP parm_blk,CRN crn)
{
   GC_PARM_DATA *parmp = NULL;
   parmp = gc_util_next_parm(parm_blk,parmp);

   if (!parmp)
   {
      return GC_ERROR;
   }

   while (NULL != parmp)
   {
      switch (parmp->set_ID)
      {
         case IPSET_TUNNELEDSIGNALMSG:
            switch (parmp->parm_ID)
            {
               case IPPARM_TUNNELEDSIGNALMSG_CONTENT:
                  printf("\tReceived extension data (TSM) Msg Content: %s\n",
                         parmp->value_buf);
                  break;
               case IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJID:
                  printf("\tReceived extension data (TSM) PROTOCOL_OBJID: %s\n",
                         parmp->value_buf);
                  break;

               case IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID:
               {
                  if(parmp->value_size == sizeof(IP_TUNNELPROTOCOL_ALTID))
                  {
                     IP_TUNNELPROTOCOL_ALTID *ptsmTpAltId;
                     ptsmTpAltId = (IP_TUNNELPROTOCOL_ALTID *)(&(parmp->value_buf));
                     printf("\tReceived extension data (TSM) Protocol Alt id:
                            Type=%s, Variant=%s, Sub Id=%s\n",
                            ptsmTpAltId->protocolType,
                            ptsmTpAltId->protocolVariant,
                            ptsmTpAltId->subIdentifier);
                  }
               }
               break;

               case IPPARM_TUNNELEDSIGNALMSG_NSDATA_DATA:
                  printf("\tReceived extension data (TSM NSDATA) DATA: %s\n",
                         parmp->value_buf);
                  break;

               case IPPARM_TUNNELEDSIGNALMSG_NSDATA_OBJID:
                  printf("\tReceived extension data (TSM NSDATA) OBJID: %s\n",
                         parmp->value_buf);
                  break;

               case IPPARM_TUNNELEDSIGNALMSG_NSDATA_H221NS:
               {
                  if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
                  {
                     IP_H221NONSTANDARD *pH221NonStandard;
                     pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
                     printf("\tReceived extension data (NSDATA) h221:CC=%d, Ext=%d, MC=%d\n",
                            pH221NonStandard->country_code,
                            pH221NonStandard->extension,
                            pH221NonStandard->manufacturer_code);
                  }
               }
               break;

         default:
            printf("\tReceived unknown (TSM NSDATA) extension parmID %d\n",
                   parmp->parm_ID);
            break;
      }
      break;

   parmp = gc_util_next_parm(parm_blk,parmp);
   }

}
```

4.21 Specifying RTP Stream Establishment

When using Global Call, RTP streaming can be established before the call is connected (that is, before the calling party receives the GCEV_CONNECTED event). This feature enables a voice message to be played to the calling party (for example, a message stating that the called party is unavailable for some reason) without the calling party being billed for the call.

The **gc_SetUserInfo( )** function can be used to specify call-related information such as coder information and display information before issuing **gc_CallAck( )**, **gc_AcceptCall( )** or **gc_AnswerCall( )**. See Section 7.3.25, "gc_SetUserInfo( ) Variances for IP", on page 337 for more information.

On the called party side, RTP streaming can be established before any of the following functions are issued to process the call:

- **gc_AcceptCall( )** – SIP Ringing (180) message returned to the calling party
- **gc_AnswerCall( )** – SIP OK (200) message returned to the calling party

4.22 Managing Quality of Service Alarms

Global Call supports the setting and retrieving of Quality of Service (QoS) thresholds and the handling of a QoS alarm when it occurs. The QoS thresholds supported by Global Call are:

- jitter
- lost packets
- RTCP inactivity
- RTP inactivity

When using Global Call with other technologies (such as E1 CAS or T1 Robbed Bit), alarms are managed and reported on the network device. For example, when **gc_OpenEx( )** is issued, specifying both a network device (dtiB1T1) and a voice device (dxxxB1C1) in the **devicename** parameter, the function retrieves a Global Call line device. This Global Call line device can be used directly in Global Call Alarm Management System (GCAMS) functions to manage alarms on the network device.

When using Global Call with IP technology, alarms such as QoS alarms are more directly related to the media processing and are therefore reported on the media device rather than on the network device. When **gc_OpenEx( )** is issued, specifying both a network device (iptB1T1) and a media device (ipmB1C1) in the **devicename** parameter, two Global Call line devices are created:

- The first Global Call line device corresponds to the network device and is retrieved in the **gc_OpenEx( )** function.
- The second Global Call line device corresponds to the media device and is retrieved using the **gc_GetResourceH( )** function. This is the line device that must be used with GCAMS functions to manage QoS alarms. See the *Global Call API Programming Guide* for more information about GCAMS.

Note: Applications **must** include the *gcipmlib.h* header file before Global Call can be used to set or retrieve QoS threshold values.

4.22.1 Alarm Source Object Name

In Global Call, alarms are managed using the Global Call Alarm Management System (GCAMS). Each alarm source is represented by an Alarm Source Object (ASO) that has an associated name.

When using Global Call with IP, the ASO name is **IPM QoS ASO**. The ASO name is useful in many contexts, for example, when configuring a device for alarm notification.

4.22.2 Retrieving the Media Device Handle

To retrieve the Global Call line device corresponding to the media device, use the **gc_GetResourceH( )** function. See Section 7.3.11, "gc_GetResourceH( ) Variances for IP", on page 306 for more information.

The Global Call line device corresponding to the media device is the device that must be used with GCAMS functions to manage QoS alarms.

4.22.3 Setting QoS Threshold Values

To set QoS threshold values, use the **gc_SetAlarmParm( )** function. See Section 7.3.23, "gc_SetAlarmParm( ) Variances for IP", on page 333 for more information.

The following code demonstrates how to set QoS threshold values.

Note: The following code uses the IPM_QOS_THRESHOLD_INFO structure from the IP Media Library (IPML). See the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information.

```
/*****************************************************************************
Routine: SetAlarmParm
Assumptions/Warnings: None.
Description: calls gc_SetAlarmParm()
Parameters: handle of the Media device
Returns: None
******************************************************************************/

void SetAlarmParm(int hMediaDevice)
{
   ALARM_PARM_LIST alarm_parm_list;
   IPM_QOS_THRESHOLD_INFO QoS_info;
   alarm_parm_list.n_parms = 1;
   QoS_info.unCount=1;
   QoS_info.QoSThresholdData[0].eQoSType = QOSTYPE_JITTER;
   QoS_info.QoSThresholdData[0].unTimeInterval = 1000;
   QoS_info.QoSThresholdData[0].unDebounceOn = 5000;
   QoS_info.QoSThresholdData[0].unDebounceOff = 15000;
   QoS_info.QoSThresholdData[0].unFaultThreshold = 50;
   QoS_info.QoSThresholdData[0].unPercentSuccessThreshold = 90;
   QoS_info.QoSThresholdData[0].unPercentFailThreshold = 10;

   alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.pstruct =
   (void *) &QoS_info;

   if (gc_SetAlarmParm(hMediaDevice, ALARM_SOURCE_ID_NETWORK_ID,
        ParmSetID_qosthreshold_alarm, &alarm_parm_list, EV_SYNC)!= GC_SUCCESS)
   {
      /* handle gc_SetAlarmParm() failure */
      printf("SetAlarmParm(hMediaDevice=%d, mode=EV_SYNC) Failed", hMediaDevice);
      return;
   }
   printf("SetAlarmParm(hMediaDevice=%d, mode=EV_SYNC) Succeeded", hMediaDevice);
}
```

4.22.4 Retrieving QoS Threshold Values

To retrieve QoS threshold values, use the **gc_GetAlarmParm( )** function. See Section 7.3.8, "gc_GetAlarmParm( ) Variances for IP", on page 302 for more information.

The following code demonstrates how to retrieve QoS threshold values.

Note: The following code uses the IPM_QOS_THRESHOLD_INFO structure from the IP Media Library (IPML). See the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information.

```
/*****************************************************************************
Routine: GetAlarmParm
Assumptions/Warnings: None
Description: calls gc_GetAlarmParm()
Parameters: handle of Media device
Returns: None
******************************************************************************/

void GetAlarmParm(int hMediaDevice)
{
   ALARM_PARM_LIST alarm_parm_list;
   unsigned int n;
   IPM_QOS_THRESHOLD_INFO QoS_info;
   IPM_QOS_THRESHOLD_INFO *QoS_infop;

   QoS_info.unCount=2;
   QoS_info.QoSThresholdData[0].eQoSType = QOSTYPE_LOSTPACKETS;
   QoS_info.QoSThresholdData[1].eQoSType = QOSTYPE_JITTER;

   /* get QoS thresholds for LOSTPACKETS and JITTER */
   alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.pstruct = (void *) &QoS_info;
   alarm_parm_list.n_parms = 1;

   if (gc_GetAlarmParm(hMediaDevice, ALARM_SOURCE_ID_NETWORK_ID,
        ParmSetID_qosthreshold_alarm, &alarm_parm_list, EV_SYNC) != GC_SUCCESS)
   {
      /* handle gc_GetAlarmParm() failure */
      printf("gc_GetAlarmParm(hMediaDevice=%d, mode=EV_SYNC) Failed", hMediaDevice);
      return;
   }

   /* display threshold values retrieved */
   printf("n_parms = %d\n", alarm_parm_list.n_parms);
   QoS_infop = alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.pstruct;
   for (n=0; n < QoS_info.unCount; n++)
   {
      printf("QoS type = %d\n", QoS_infop->QoSThresholdData[n].eQoSType);
      printf("\tTime Interval = %u\n", QoS_infop->QoSThresholdData[n].unTimeInterval);
      printf("\tDebounce On = %u\n", QoS_infop->QoSThresholdData[n].unDebounceOn);
      printf("\tDebounce Off = %u\n", QoS_infop->QoSThresholdData[n].unDebounceOff);
      printf("\tFault Threshold = %u\n", QoS_infop->QoSThresholdData[n].unFaultThreshold);
      printf("\tPercent Success Threshold = %u\n",
             QoS_infop->QoSThresholdData[n].unPercentSuccessThreshold);
      printf("\tPercent Fail Threshold = %u\n",
             QoS_infop->QoSThresholdData[n].unPercentFailThreshold);
      printf("\n\n");
   }
}
```

4.22.5 Handling QoS Alarms

The application must first be enabled to receive notification of alarms on the specified line device. The following code demonstrates how this is achieved.

```
/****************************************************************
*        NAME: enable_alarm_notification(struct channel *pline)
* DESCRIPTION: Enables all alarms notification for pline
*              Also fills in pline->mediah
*       INPUT: pline - pointer to channel data structure
*     RETURNS: None - exits if error
*    CAUTIONS: Does no sanity checking as to whether or not the technology
*              supports alarms - assumes caller has done that already
***************************************************************/

static void enable_alarm_notification(struct channel *pline)
{
   char     str[MAX_STRING_SIZE];
   int      alarm_ldev;                /* linedevice that alarms come on */

   alarm_ldev = pline->ldev;           /* until proven otherwise */

   if (pline->techtype == H323)
   {
      /* Recall that the alarms for IP come on the media device, not the network device */
      if (gc_GetResourceH(pline->ldev, &alarm_ldev, GC_MEDIADEVICE) != GC_SUCCESS)
      {
         sprintf(str, "gc_GetResourceH(linedev=%ld, &alarm_ldev,
                 GC_MEDIADEVICE) Failed", pline->ldev);
         printandlog(pline->index, GC_APIERR, NULL, str);
         exitdemo(1);
      }
      sprintf(str, "gc_GetResourceH(linedev=%ld, &alarm_ldev,
              GC_MEDIADEVICE) passed, mediah = %d", pline->ldev, alarm_ldev);
      printandlog(pline->index, MISC, NULL, str);
      pline->mediah = alarm_ldev;         /* save for later use */
   }
   else
   {
      printandlog(pline->index, MISC, NULL, "Not setting pline->mediah
                  since techtype != H323");
   }
   sprintf(str, "enable_alarm_notification - pline->mediah = %d\n", (int) pline->mediah);

   if (gc_SetAlarmNotifyAll(alarm_ldev, ALARM_SOURCE_ID_NETWORK_ID,
       ALARM_NOTIFY) != GC_SUCCESS)
   {
      sprintf(str, "gc_SetAlarmNotifyAll(linedev=%ld,
              ALARM_SOURCE_ID_NETWORK_ID, ALARM_NOTIFY) Failed", pline->ldev);
      printandlog(pline->index, GC_APIERR, NULL, str);
       exitdemo(1);
   }
   sprintf(str, "gc_SetAlarmNotifyAll(linedev=%ld, ALARM_SOURCE_ID_NETWORK_ID,
           ALARM_NOTIFY) PASSED", pline->ldev);
   printandlog(pline->index, MISC, NULL, str);
}
```

When a GCEV_ALARM event occurs, use the Global Call Alarm Management System (GCAMS) functions such as, **gc_AlarmNumber( )** to retrieve information about the alarm. The following code demonstrates how to process a QoS alarm when it occurs. In this case the application simply logs information about the alarm.

```
/****************************************************************
*        NAME: void print_alarm_info(METAEVENTP metaeventp,
*                                    struct channel *pline)
* DESCRIPTION: Prints alarm information
*      INPUTS: metaeventp - pointer to the alarm event
*              pline - pointer to the channel data structure
*     RETURNS: NA
*    CAUTIONS: Assumes already known to be an alarm event
***************************************************************/

static void print_alarm_info(METAEVENTP metaeventp, struct channel *pline)
{
   long              alarm_number;
   char              *alarm_name;
   unsigned long     alarm_source_objectID;
   char              *alarm_source_object_name;
   char              str[MAX_STRING_SIZE];

   if (gc_AlarmNumber(metaeventp, &alarm_number) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmNumber(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   if (gc_AlarmName(metaeventp, &alarm_name) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmName(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   if (gc_AlarmSourceObjectID(metaeventp, &alarm_source_objectID) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmSourceObjectID(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   if (gc_AlarmSourceObjectName(metaeventp, &alarm_source_object_name) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmSourceObjectName(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   sprintf(str, "Alarm %s (%d) occurred on ASO %s (%d)",
           alarm_name, (int) alarm_number, alarm_source_object_name,
           (int) alarm_source_objectID);

   printandlog(pline->index, MISC, NULL, str);
}
```

See the *Global Call API Programming Guide* for more information about the operation of GCAMS and the *Global Call API Library Reference* for more information about GCAMS functions.

4.23 Registration

In an H.323 network, a Gatekeeper manages the entities in a specific zone and an endpoint must register with the Gatekeeper to become part of that zone. In a SIP network, a Registrar manages a set of associations or bindings between Addresses-of-Record and actual endpoint addresses for a domain. Global Call provides applications with the ability to perform endpoint registration. These capabilities are described in the following topics:

- Registration Overview
- Registration Operations
- Sending and Receiving Nonstandard Registration Messages (H.323)
- Code Examples
- Gatekeeper Registration Failure (H.323)

4.23.1 Registration Overview

When using Global Call to perform endpoint registration, the following conditions and restrictions apply:

- An application must use an IPT board device handle to perform registration. A board device handle can be obtained by using **gc_OpenEx( )** with a **devicename** parameter of “N_iptBx”.
- When using the H.323 protocol, an application must perform registration before using **gc_OpenEx( )** on any other line device.
- Once an H.323 application chooses to be registered with a Gatekeeper, it can change its Gatekeeper by deregistering and reregistering with another Gatekeeper, but it cannot handle calls without being registered with some Gatekeeper.
- Once an H.323 application is registered and has active calls, deregistration or switching to a different Gatekeeper must be done only when all calls are in the Idle state. The **gc_ResetLineDev( )** function can be used to put channels in the Idle state.
- When using the **gc_ReqService( )** function, two mandatory parameters IDs, PARM_REQTYPE and PARM_ACK, both in the GCSET_SERVREQ parameter set, are required in the GC_PARM_BLK parameter block. These parameters are required by the generic service request mechanism provided by Global Call and are not sent in any registration message.
- When setting H.323 alias or SIP Transport Address information, the **gc_ReqService( )** function can include more than one address in the GC_PARM_BLK associated with the function. Prefixes are ignored for SIP.
- Registration operations cannot be included in the preset registration information using **gc_SetConfigData( )**.

Note: When using SIP, it is important to note that RFC3261 specifies that the “host” portion of a URI that is given as a numeric IPv4 address (for example, 123.211.40.90) and one given as a domain name (for example, example.com) are treated as unique even if they actually resolve to the same entity. Applications should be careful to use the same URI that was used in the initial registration for all subsequent registration operations.

Global Call provides a number of options for registration and manipulation of registration information. The Global Call API simplifies and abstracts the network RAS messages in H.323 and REGISTER messages in SIP.

H.323

In H.323, the following functionality (and the corresponding RAS messages) is supported:

- locating a gatekeeper via unicast or multicast (RAS messages: GRQ/GCF/GRJ)
- registration (RAS message: RRQ)
- specifying one-time or periodical registration (RAS message: RRQ)
- changing registered information (RAS message: RRQ)
- removing registered information by value (RAS message: RRQ)
- sending non-standard registration message (RAS message: NonStandardMessage)
- deregistering (RAS messages: URQ/UCF/URJ)
- handling calls according to the gatekeeper policy for directing and routing calls (RAS messages: ARQ/ACF/ARJ, DRQ/DCF/DRJ)

Note: For detailed information on RAS negotiation, see *ITU-T Recommendation H.225.0*.

SIP REGISTER Method

The SIP REGISTER method is used to register associations between a media endpoint alias and its real (transport) address. The associations are maintained in a SIP Registrar and are used for SIP call routing. Global Call only supports registering, de-registering, and querying with a Registrar; it does not support receiving SIP REGISTER requests. Table 16 associates abstract Registrar registration concepts with SIP REGISTER message elements and Global Call interface elements.

Table 16. SIP REGISTER Method

Concept	SIP REGISTER Element	Global Call Interface Element
Initiate registration	REGISTER method	gc_ReqService( )
Registrar’s address	Request-URI	IPSET_REG_INFO IPPARM_REG_ADDRESS IP_REGISTER_ADDRESS.reg_server
Alias (Address-of-record)	To header field	IPSET_REG_INFO IPPARM_REG_ADDRESS IP_REGISTER_ADDRESS.reg_client
Sender's address-of-record (only used in 3rd party call control environments)	From header field	IPSET_SIP_MSGINFO IPPARM_FROM (string) †
† If not supplied by application, library automatically uses the value provided for Alias		

Table 16. SIP REGISTER Method (Continued)

Concept	SIP REGISTER Element	Global Call Interface Element
Transport address (actual endpoint address)	Contact header field	IPSET_LOCAL_ALIAS IPPARM_ADDRESS_TRANSPARENT (string)
Auto-refresh interval	Expires header field	IPSET_REG_INFO IPPARM_REG_ADDRESS IP_REGISTER_ADDRESS.time_to_live
† If not supplied by application, library automatically uses the value provided for Alias		

Note: Because the Transport Address is sent to the Registrar in the Contact header field, it must include a valid URI scheme prefix, namely “sip:” or “sips:”. If the application does not supply a scheme prefix, the call control library automatically inserts “sip:”, but only after the SIP stack has generated a parser error. These stack parser errors are written to the RTFLog file unless the user turns off logging of this type of error. To turn off the logging of these parser errors, find the line

```
<MClient name="PARSER" state = "1"/>
```

in the *RtfConfigWin.xml* file and replace it with

```
<MClient name="PARSER" state = "1">
   <MClientLabel name="Error" state = "0"/>
</MClient>
```

4.23.2 Registration Operations

Applications perform all types of registration operations (registering, deregistering, querying, and modifiying or deleting registration information) using the **gc_ReqService( )** function. The specific operation to perform, and the information necessary for that operation are included in the parameter elements in a GC_PARM_BLK that is passed to the **gc_ReqService( )** function. The specific parameters to use for each type of operation are described inthe following subsections.

In addition to the parameter elements that are required for H.323 or SIP registrations, there are two two mandatory parameter that are required by the generic service request mechanism even though they have no meaning in the context of H.323/SIP endpoint registration. These two parameters, GCSET_SERVREQ / PARM_REQTYPE and GCSET_SERVREQ / PARM_ACK, must always be present in the GC_PARM_BLK.

The **gc_ReqService( )** function operates in the asynchronous mode, and the application receives a GCEV_SERVICERESP termination event if the call control library succeeds in communicating with the registration server. It is important to note that a GCEV_SERVICERESP event indicates that the requested registration operation was completed successfully only if the event’s result code (the ccValue field in the GC_INFO structure from a **gc_ResultInfo( )** function call) is IPERR_OK. If the result code is any other value, there was some sort of error during the registration.

4.23.2.1 Configuring the Maximum Number of Registrations (SIP)

Because internal stack resources are required to monitor each unique binding that is set to auto-refresh, and because auto-refresh is the default mode for SIP registration, the Global Call call control library allows the application to configure the maximum number of registrations when each virtual board is started. This configuration is accomplished via the sip_registrar_registrations field in the IP_VIRTBOARD structure that is used when starting a given virtual board. The default value for this field sets the maximum number of registrations to be the same as the maximum number of SIP calls (the sip_max_calls field in IP_VIRTBOARD), which is appropriate in most situations. If the application needs to register all or most users with more than one Registrar, or to register multiple transport addresses for all or most users, it needs to increase this configuration parameter from the default value.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions, which must be called before **gc_Start( )**, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The following code snippet illustrates how an application might increase the maximum number of registrations on the second board to allow two registrations per user:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[1].sip_registrar_registrations = 240; /* override defaults no. of registrations*/
```

If an application requests a registration that exceeds the configured maximum number of registrations for the virtual board, the application's request is rejected by the call control library, which generates a GCEV_SERVICERESP event with the response code IPEC_REG_FAIL_insufficientInternalResources.

4.23.2.2 Locating a Registration Server

A Global Call application can choose to use a known address for the registration server (H.323 Gatekeeper or SIP Registrar) or to discover a registration server by multicasting to a well-known address on which registration servers listen. This choice is determined by the IP address specified as the registration address during registration.

The registration address is specified in the IPPARM_REG_ADDRESS parameter in the IPSET_REG_INFO parameter set. The value of the IPPARM_REG_ADDRESS is an IP_REGISTER_ADDRESS structure, which includes a reg_server field that contains the address value. A specific range of IP addresses is reserved for multicast transmission:

- If the application specifies an address in the range of multicast addresses or specifies the default multicast address (IP_REG_MULTICAST_DEFAULT_ADDR), then registration server discovery is selected.
- If the application specifies an address outside the range of multicast addresses, then registration with a specific server is selected.

Note: In SIP, if the reg_server field contains NULL or an invalid address, the default multicast address is automatically used by the library.

When using the default multicast registration address, the application can specify the maximum number of hops (connections between routers) in the max_hops field of the IP_REGISTER_ADDRESS structure.

H.323

For H.323 registration, the port number used for RAS is one less than the port number used for signaling. To avoid a port conflict when configuring multiple devices, do not assign consecutive H.323 signaling port numbers to devices in the IPCCLIB_START_DATA structure. See Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340 for more information.

4.23.2.3 Registration Requests

An application uses the **gc_ReqService( )** function to register with a Gatekeeper/Registrar. The registration information in this case is included in the GC_PARM_BLK associated with the **gc_ReqService( )** function. See Section 4.23.4, “Code Examples”, on page 229 for more information.

H.323

If registration is initiated by a Global Call application via **gc_ReqService( )** and the Gatekeeper rejects the registration, a GCEV_SERVICERESP event containing the result code IPEC_RASReasonInvalidIPEC_RASAddress.

SIP

In SIP, an application can make multiple simultaneous registration requests to different Registrars or to the same Registrar on behalf of different User Agents. To allow the application to distinguish among multiple completion events from these simultaneous requests, the data associated with the completion event contains a Service ID parameter that is the number that was handed back to the application when the initiating **gc_ReqService( )** was made.

According to RFC3261, applications may not make more than one registration attempt at the same time for a particular User Agent on a particular Registrar. If the application attempts to send a second REGISTER request to a given Registrar for the same UA before the initial REGISTER transaction completes, the call control library rejects the request and generates a GCEV_SERVICERESP event containing the result code IPEC_REG_FAIL_registrationTransactionInProgress to notify the application of the rejection.

4.23.2.4 Auto-Refreshing Registrations

Global Call enables an application to specify a one-time registration or periodic registration where bindings are automatically re-registered with the Gatekeeper/Registrar at the interval (in seconds) specified by the application. Applications that are using automatic re-registration are not notified of successful registration refresh transactions.

H.323

In H.323 registration, periodic registration is achieved by setting the time_to_live field in the IP_REGISTER_ADDRESS structure. If the parameter is set to zero (the default value), then the stack uses one-time registration functionality. If the parameter is set to a value greater than zero, then each registration with the server is valid for the specified number of seconds and the stack automatically refreshes its request before timeout.

If the Gatekeeper rejects the registration (sends RRJ) during periodic registration, the application will receive an unsolicited GCEV_TASKFAIL event that contains a reason provided by the Gatekeeper. If the Gatekeeper does not set the reason, the default reason is IPEC_RASReasonInvalidIPEC_RASAddress.

SIP

When using SIP, auto-refresh is used by default. If the application does not explicitly set the time_to_live value in the IP_REGISTER_ADDRESS structure (that is, doesn't change the value from its default value of 0), the call control library automatically sets the Expires header field in the REGISTER request to a a value of 3600 seconds. If the application wishes to request a longer or shorter auto-refresh interval, it simply sets the time_to_live field to the appropriate value, and that value is set in the Expires header field.

The actual expiration time for registration is determined by the Registrar, which may or may not accept the Expires value suggested in the REGISTER request. The expiration time received from the Registrar is recorded and used by the Global Call library only if the application has not disabled the auto-refresh mechanism. If the expiration time returned by the Registrar is greater than 40 seconds, re-registration is attempted 30 seconds before the registration is set to expire. If the expiration time returned by the Registrar is 40 seconds or less, re-registration is attempted within 5 seconds of receiving that response. When auto-refresh is enabled, the call control library rejects registration refresh times of 5 seconds or less and generates a GCEV_SERVICERESP event with the response code IPEC_REG_FAIL_invalidExpires. If a refresh time of 5 seconds or less is actually desired, the application must disable the auto-refresh mechanism for each binding and will then be responsible for explicitly renewing those bindings with the Registrar.

If the automatic re-registration fails because the Registrar rejects the request, the Registrar's response code is forwarded to the application in a GCEV_SERVICERESP event. Automatic re-registration can also fail if constant application activity on a particular binding causes re-registration to be postponed beyond the binding's actual expiration time. (A 500ms postponement occurs when an auto re-registration attempt collides with a current application transaction on the same binding.) In this case the GCEV_SERVICERESP event sent to the application contains the result code IPEC_REG_FAIL_reRegistrationRequired. In either case, the application is then responsible for re-registering the binding, if appropriate.

The extra data associated with a re-registration failure event includes:

- Request-URI (as IPSET_SIP_MSGINFO / IPPARM_REQUEST_URI)
- To header field value (as IPSET_SIP_MSGINFO / IPPARM_TO)
- From header field value, if one had been provided (as IPSET_SIP_MSGINFO / IPPARM_TO)
- Contact header field value that failed to auto refresh (as IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT)

A SIP application can explicitly disable or re-enable auto-refresh on a per registration basis, by using the following parameter element:

IPSET_REG_INFO
  IPPARM_REG_AUTOREFRESH
  and one of the following values:

- IP_AUTOREFRESH_DISABLE – disable auto-refresh for a specific registration
- IP_AUTOREFRESH_ENABLE – enable auto-refresh for a specific registration, using the non-zero value specified in IP_REGISTER_ADDRESS.time_to_live or the default value of 3600 in the Expires header field

Note: If this parameter is not present in the GC_PARM_BLK when registration is requested, auto-refresh is enabled by default.

4.23.2.5 Receiving Notification of Registration

An application that sends a registration request to a Gatekeeper/Registrar receive notification of whether the registration is successful or not. When using Global Call the application receives a GCEV_SERVICERESP termination event with an associated GC_PARM_BLK that contains the following elements:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  with one of the following values:

- IP_PROTOCOL_H323
- IP_PROTOCOL_SIP

IPSET_REG_INFO
  IPPARM_REG_STATUS
  with one of the following values:

- IP_REG_CONFIRMED – registration operation completed properly
- IP_REG_REJECTED – registration operation did not complete properly; the **gc_ResultInfo( )** function can be used to retrieve the reason for the failure

SIP

For registrations with a SIP Registrar, the GC_PARM_BLK associated with the GCEV_SERVICERESP termination event also contains the following element:

IPSET_REG_INFO
  IPPARM_REG_SERVICE

- value = the Service ID that was handed back to the application when the initiating **gc_ReqService( )** was made

This Service ID can be used by the application to distinguish among multiple events returned on a given handle, since the application can send multiple simultaneous REGISTER requests to different Registrars or to the same Registrar on behalf of different User Agents.

4.23.2.6 Querying Registration Information (SIP)

Global Call provides a mechanism for a SIP application to query a Registrar to determine what bindings currently exist. To do this, the application calls **gc_ReqService( )** with the following parameter element included in the GC_PARM_BLK that is passed to the function:

IPSET_REG_INFO
 IPPARM_OPERATION_REGISTER
- value = IP_REG_QUERY_INFO

The application specifies the Registrar and Alias to query by including the following parameter element in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
 IPPARM_REG_ADDRESS
- value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias to query

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

By default, the registration query operation returns all Transport Addresses that are currently registered for the specified Alias by the application. If the application wishes to query *all* Transport Addresses that have been registered in the Registrar for the specified Alias (that is, all registrations by all applications), the GC_PARM_BLK that it supplies to the **gc_ReqService( )** function must include the following element:

IPSET_LOCAL_ALIAS
 IPPARM_ADDRESS_TRANSPARENT
- value = "*"

The GCEV_SERVICERESP completion event for this function call contains all current bindings for the specified Address of Record in a series of IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT parameter elements. The value of each of these elements is a null-terminated string that contains a current binding created by this application along with any header field parameters that were appended by the Registrar.

4.23.2.7 Changing Registration Information

Global Call provides the ability to modify or add to the registration information after it has been registered with the Gatekeeper/Registrar. To change registration information, the application uses the **gc_ReqService( )** function and passes a GC_PARM_BLK that contains the following element:

IPSET_REG_INFO
 IPPARM_OPERATION_REGISTER
 and one of the following values:
- IP_REG_SET_INFO – override existing registration
- IP_REG_ADD_INFO – add to existing registration information

A SIP application can specify the Registrar and Alias to modify information for by including the following parameter in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
  IPPARM_REG_ADDRESS
- value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

The overriding or additional information is contained in other elements in the GC_PARM_BLK. The elements that can be included are given in Table 29, “Registration Information When Using H.323”, on page 331 and Table 30, “Registration Information When Using SIP”, on page 332.

Note: For SIP, the Sender’s Address of Record that was used to initially register a binding never changes. Any attempt to update this value is ignored.

4.23.2.8 Removing Registered Information by Value

Global Call allows applications to delete one or more registration values from an existing registration. This applies to aliases and supported prefixes in H.323, and to Transport Addresses in SIP. When an application needs to delete one or more specific values, it uses the **gc_ReqService( )** function and passes a GC_PARM_BLK that contain the following parameter element:

IPSET_REG_INFO
  IPPARM_OPERATION_REGISTER
- value = IP_REG_DELETE_BY_VALUE

Each H.323 alias or SIP Transport Address to be deleted is contained in an additional element in the GC_PARM_BLK that uses the IPSET_LOCAL_ALIAS set ID and the appropriate parameter ID for the address type.

H.323

Supported prefixes to be deleted from the registration are specified via GC_PARM_BLK elements that use the IPSET_SUPPORTED_PREFIXES set ID.

If the string that is contained in the value of the GC_PARM_BLK element matches a registered alias or supported prefix, it is deleted from the local database and an updated list is sent to the Gatekeeper.

SIP

A SIP application can specify the Registrar and Alias to modify information for by including the following parameter in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
  IPPARM_REG_ADDRESS
- value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

If the GC_PARM_BLK does not contain any IPSET_LOCAL_ALIAS elements specifying Transport Addresses to be deleted, no bindings will be deleted and the function call has the same result as the query operation described in Section 4.23.2.6, "Querying Registration Information (SIP)", on page 226.

If the GC_PARM_BLK contains an IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT parameter element with the value `"*"`, all bindings that exist in the specified Registrar for the specified Alias are deleted, regardless of what application created them.

4.23.2.9 Deregistering

Global Call provides the ability to deregister from a Gatekeeper/Registrar. When deregistering, the application can decide whether to keep the registration information locally or delete it. To deregister, an application uses the **gc_ReqService( )** function and passes it a GC_PARM_BLK that contains the following element:

IPSET_REG_INFO
  IPPARM_OPERATION_DEREGISTER
  and one of the following values:
- IP_REG_MAINTAIN_LOCAL_INFO – keep the registration information locally
- IP_REG_DELETE_ALL – delete the local registration information

See Section 4.23.4.2, "Deregistration Example", on page 231 for more information.

SIP

A SIP application can specify the Registrar and Alias to deregister by including the following parameter in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
  IPPARM_REG_ADDRESS
- value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

If the GC_PARM_BLK does not contain any IPSET_LOCAL_ALIAS elements specifying Transport Addresses to be deleted, all bindings previously created by this application for the specified Alias will be removed from the Registrar.

If the GC_PARM_BLK contains an IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT parameter element with the value `"*"`, all bindings that exist in the specified Registrar for the specified Alias are deleted, regardless of what application created them.

4.23.3 Sending and Receiving Nonstandard Registration Messages (H.323)

Global Call provides the ability to send nonstandard messages to and receive nonstandard messages from the gatekeeper or registrar. To send nonstandard messages, the application uses the **gc_Extension( )** function. The first element must be set as described in Section 8.2.15, "IPSET_MSG_REGISTRATION", on page 366. Other elements are set as in conventional nonstandard messages; see Section 8.2.18, "IPSET_NONSTANDARDDATA", on page 368.

An unsolicited GCEV_EXTENSION event with an extension ID (ext_id) of IPEXTID_RECEIVEMSG can be received that contains a nonstandard registration message. The associated GC_PARM_BLK contains the message details as follows:

- A message identifier element that contains the IPSET_MSG_REGISTRATION parameter set ID and an IPPARM_MSGTYPE parameter ID with a value of IP_MSGTYPE_REG_NONSTD.
- One or more additional elements that contain the message data of the form:
  - IPSET_NONSTANDARDDATA with
    - IPPARM_NONSTANDARDDATA_DATA – the maximum length is MAX_NS_PARM_DATA_LENGTH (128)
    - IPPARM_NONSTANDARDDATA_OBJID – the maximum length is MAX_NS_PARM_OBJID_LENGTH (40)

  OR
  - IPSET_NONSTANDARDDATA with
    - IPPARM_NONSTANDARDDATA_DATA – the maximum length is MAX_NS_PARM_DATA_LENGTH (128)
    - IPPARM_H221NONSTANDARD

4.23.4 Code Examples

4.23.4.1 Registration Example

The following code example shows how to populate a GC_PARM_DATA structure that can be used to register an endpoint with a gatekeeper (H.323) or registrar (SIP). The GC_PARM_DATA structure contains the following registration information:

- two mandatory parameters required by the generic **gc_ReqService( )** function
- the protocol type (H.323, SIP, or both)

- the type of operation (register/deregister) and sub-operation (set information, add information, delete by value, delete all)
- the IP address to be registered
- the endpoint type to register as
- a number of local aliases
- a number of supported prefixes

```
int boardRegistration(IN LINEDEV boarddev)
{
   GC_PARM_BLKP pParmBlock = NULL;
   int frc = GC_SUCCESS;

   /****** Two (mandatory) elements that are not related directly to
   the server-client negotiation ********/
   frc = gc_util_insert_parm_val(&pParmBlock,
                                 GCSET_SERVREQ,
                                 PARM_REQTYPE,
                                 sizeof(char),
                                 IP_REQTYPE_REGISTRATION);

   frc = gc_util_insert_parm_val(&pParmBlock,
                                 GCSET_SERVREQ,
                                 PARM_ACK,
                                 sizeof(char),
                                 1);

   /******Setting the protocol target***********/
   frc = gc_util_insert_parm_val(&pParmBlock,
                                 IPSET_PROTOCOL,
                                 IPPARM_PROTOCOL_BITMASK,
                                 sizeof(char),
                                 IP_PROTOCOL_H323); /*can be H323, SIP or Both*/

   /****** Setting the operation to perform ***********/
   frc = gc_util_insert_parm_val(&pParmBlock,
                                 IPSET_REG_INFO,
                                 IPPARM_OPERATION_REGISTER, /* can be Register or Deregister */
                                 sizeof(char),
                                 IP_REG_SET_INFO); /* can be other relevant "sub" operations */

   /****** Setting address information ***********/
   IP_REGISTER_ADDRESS registerAddress;
   strcpy(registerAddress.reg_server,"101.102.103.104"); /* set server address*/
   strcpy(registerAddress.reg_client,"user@10.20.30.40");   /* set alias for SIP*/
   registerAddress.max_hops = regMulticastHops;
   registerAddress.time_to_live = regTimeToLive;

   frc = gc_util_insert_parm_ref(&pParmBlock,
                                 IPSET_REG_INFO,
                                 IPPARM_REG_ADDRESS,
                                 (UINT8)sizeof(IP_REGISTER_ADDRESS),
                                 &registerAddress);

   /****** Setting endpoint type to GATEWAY (H.323 only) ***********/
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_REG_INFO,
                           IPPARM_REG_TYPE,
                           (unsigned char)sizeof(EPType),
                           IP_REG_GATEWAY);
   /**** Setting terminalAlias information ****/
   /**** May repeat this line with different addresses and address types ****/
   frc = gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_LOCAL_ALIAS,
                              (unsigned short)IPPARM_ADDRESS_EMAIL,
                              (UINT8)(strlen("someone@someplace.com")+1),
                              "someone@someplace.com");

   /****** Setting supportedPrefixes information ***********/
   /**** With H.323 - may repeat this line with different supported prefixes and
           supported prefix types ****/
   /**** SIP does not allow setting of this parm block ****/
   frc = gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_SUPPORTED_PREFIXES,
                              (unsigned short)IPPARM_ADDRESS_PHONE,
                              (UINT8)(strlen("011972")+1),
                              "011972");

   /****** Send the request  ***********/
   unsigned long   serviceID ;
   int rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                       boarddev,
                       &serviceID,
                       pParmBlock,
                       NULL,
                       EV_ASYNC);

   if (rc != GC_SUCCESS)
   {
      printf("failed in gc_ReqService\n");
      return GC_ERROR;
   }

   gc_util_delete_parm_blk(pParmBlock);
   return GC_SUCCESS;
}
```

4.23.4.2 Deregistration Example

The following code example shows how to populate a GC_PARM_DATA structure that can be used to deregister an endpoint with a gatekeeper (H.323). The GC_PARM_DATA structure contains the following deregistration information:

- the type of operation (in this case, deregister) and sub-operation (do not retain the registration information locally)
- two mandatory parameters required by the generic **gc_ReqService**( ) function
- the protocol type (in this case, H.323)

```
void unregister()
{
   GC_PARM_BLKP       pParmBlock = NULL;
   unsigned long      serviceID = 1;
   int                rc,frc;
   int gc_error;          // GC error code
   int cclibid;           // Call Control library ID for gc_ErrorValue
   long cc_error;         // Call Controll library error code
   char *resultmsg;       // String associated with cause code
   char *lib_name;        // Library name for cclibid
```

```
gc_util_insert_parm_val(&pParmBlock,
                        IPSET_REG_INFO,
                        IPPARM_OPERATION_DEREGISTER,
                        sizeof(unsigned char),
                        IP_REG_DELETE_ALL);

frc = gc_util_insert_parm_val(&pParmBlock,
                              GCSET_SERVREQ,
                              PARM_REQTYPE,
                              sizeof(unsigned char),
                              IP_REQTYPE_REGISTRATION);

if (frc != GC_SUCCESS)
{
   printf("failed in PARM_REQTYPE\n");
   termapp();
}

frc = gc_util_insert_parm_val(&pParmBlock,
                              GCSET_SERVREQ,
                              PARM_ACK,
                              sizeof(unsigned char),
                              IP_REQTYPE_REGISTRATION);

if (frc != GC_SUCCESS)
{
   printf("failed in PARM_ACK\n");
   termapp();
}

frc = gc_util_insert_parm_val(&pParmBlock,
                              IPSET_PROTOCOL,
                              IPPARM_PROTOCOL_BITMASK,
                              sizeof(char),
                              IP_PROTOCOL_H323); /*can be H323, SIP or Both*/

if (frc != GC_SUCCESS)
{
   printf("failed in IPSET_PROTOCOL\n");
   termapp();
}

rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                   brddev,
                   &serviceID,
                   pParmBlock,
                   NULL,
                   EV_ASYNC);

if ( GC_SUCCESS != rc)
{
   printf("gc_ReqService failed while unregestering\n");
   if (gc_ErrorValue(&gc_error, &cclibid, &cc_error) != GC_SUCCESS)
   {
      printf("gc_Start() failed:  Unable to retrieve error value\n");
   }
   else
   {
      gc_ResultMsg(LIBID_GC, (long) gc_error, &resultmsg);
      printf("gc_ReqService() failed:  gc_error=0x%X:  %s\n", gc_error, resultmsg);
      gc_ResultMsg(cclibid, cc_error, &resultmsg);
      gc_CCLibIDToName(cclibid, &lib_name);
      printf("%s library had error 0x%lx - %s\n", lib_name, cc_error, resultmsg);
   }
   gc_util_delete_parm_blk(pParmBlock);
   exit(0);
}
```

```
   printf("Unregister request to the GK was sent ...\n");
   printf("the application will not be able to make calls !!! so it will EXIT\n");
   gc_util_delete_parm_blk(pParmBlock);
   return;
}
```

4.23.5 Gatekeeper Registration Failure (H.323)

Gatekeeper registration can fail for any one of several reasons, such as disconnecting the network cable, a network topology change that result in the loss of all paths to the Gatekeeper, a Gatekeeper failure, or a Gatekeeper shutdown. Terminals may not be immediately aware of the registration failure unless a RAS registration is attempted when the cable is disconnected, in which case the transaction fails immediately because of a socket bind failure. More typically, a RAS registration failure is only detected when either the Time To Live interval (programmable, with a default of 20 seconds) or the Response timeout (2 seconds) expires. RAS failure detection times can be improved by setting the Time To Live value in the RAS registration request to a value smaller than the default value, to 10 seconds, for example.

When RAS loses the Gatekeeper registration, all existing calls are automatically disconnected by Global Call. All new calls are gracefully rejected and will continue to be rejected until RAS successfully registers or explicitly unregisters with the Gatekeeper. The application can use the **gc_ReqService( )** function to perform the re-register or unregister operation. Calls in progress that are disconnected during RAS recovery are identified by a call control library result value of IPEC_RASReasonNotRegistered in the GCEV_DISCONNECTED event.

All **gc_ReqService( )** function calls result in the return of either a GCEV_SERVICERESP (success) or GCEV_TASKFAIL (fail) completion event. If RAS registration fails (for example, as a result of an immediate socket bind failure or failure notification following a Time To Live timeout), the application receives a GCEV_TASKFAIL event. The range of applicable cause values for RAS-related GCEV_TASKFAIL events is IPEC_RASReasonMin to IPEC_RASReasonMax. The application must use the **gc_ReqService( )** function to reconfigure or register RAS in response to that event. If the RAS registration is rejected, the call control library is still cleaning up after the RAS registration failure and the application will receive another GCEV_TASKFAIL event, in which case it must issue **gc_ReqService( )** yet again.

It is recommended (but not required) that after receiving a GCEV_TASKFAIL event which identifies loss of Gatekeeper registration, the application should:

- Stop attempting to make new calls, which uses resources unnecessarily and slows down the cleanup time.
- Immediately issue a new RAS register or RAS unregister request.

RAS registration requests should be made immediately on receipt of a RAS GCEV_TASKFAIL. Recovery from the loss of registration with the gatekeeper is not completed until the call control library re-registers or attempts to unregister. Re-registration or unregistration is not attempted by the call control library until commanded by the application using the **gc_ReqService( )** function to issue a RAS REGISTER REQUEST or a RAS UNREGISTER SERVICE REQUEST respectively.

4.24 SIP Digest Authentication

Authentication is a process that allows a remote endpoint (a User Agent Server, or UAS) to verify the identity of a User Agent Client (UAC) which sends it a request. If the UAS rejects a request with a 401 or 407 response, the UAC can re-send the request in a form that includes the sender's username and password to authenticate their identity. (Once the UAC has authenticated its identity, the UAS may require further verification that the UAC is authorized to make the original request, but that is a separate process from authentication.) The standard type of SIP authentication is called "digest authentication", which refers to the encryption method used for secure transmission of the user's secret password in the message, and is documented in IETF RFC 2617.

To be able to respond automatically respond to authentication challenges, a UAC typically registers one or more triplets containing {realm, username, password}, where realm identifies the protected domain and the username and password identify the user. When a UAC receives a 401 or 407 response, it searches the triplets for a realm string that matches the one contained in the WWW-Authenticate or Proxy-Authenticate header field in the response. If it finds a matching realm string, it calculates a digest of the corresponding username and password strings and includes that result in the Authorization header field of the request it re-sends to the UAS.

The Global Call implementation of digest authorization extends this model to use quadruplets of {realm, identity, username, password}, where the identity represents the user's URI in the realm. This extension allows applications to either register a single username and password for a given realm, or multiple username/password pairs that are each associated with a different identity URI. For quadruplets that have an empty string as the identity element, the Global Call library matching process uses the realm element only, exactly as if it were using a conventional authentication triplet instead of a quadruplet. If the identity element is a non-empty string, the library compares the identity string against the URI in the From header field of the 401/407 response. When the identity is non-empty, the library re-sends the request with the username/password digest only if both the realm and identity match the appropriate fields in the response message.

As an example, if the following header fields are received in a 401 Unauthorized response:

```
From: <sip:bob@example.com>;tag=0-13c4-4129f5f4-3bf3065a-7fc2
...
WWW-Authenticate: Digest realm="atlanta.com", domain="sip:ss1.carrier.com", qop="auth",
nonce="f84f1cec41e6cbe5aea9c8e88d359", opaque="", stale=FALSE, algorithm=MD5
```

both of the following quadruplets would be considered to be matches:

{"atlanta.com", "sip:bob@example.com", "bob", "password1"}

{"atlanta.com", "", "anonymous", ""}

Applications that require multiple identities per realm set multiple quadruplets with different, non-empty identity strings. Such applications may also set a default username and password by setting a quadruplet with an empty identity string. This default username/password is only used when a 401/407 response does not match the identity in any of the triplets for the given realm and may be an anonymous authentication as shown in the preceding example.

Applications that require only a single username/password pair per realm set only a single quadruplet with an empty identity string. In this case the application would not set any quadruplets that include non-empty identity strings.

Applications that wish to use the authentication mechanism should configure the desired authentication quadruplets before calling any function that may send a SIP request. Any 401 or 407 response that is received for a request that was sent before authentication quadruplets were configured causes the call/request to be terminated and not re-sent by Global Call even if an appropriate authentication quadruplet was configured in the interim. The reason code for such a termination is IPEC_SIPReasonStatus401Unauthorized or IPEC_SIPReasonStatus407ProxyAuthenticationRequired.

Digest authentication is supported for the following SIP message types:

- BYE
- INFO (within a dialog)
- INVITE
- NOTIFY (within a dialog)
- OPTIONS (within a dialog)
- Re-INVITE
- REFER (within a dialog)
- REGISTER
- SUBSCRIBE

Authentication is specifically not supported for the following SIP message types:

- INFO (outside of a dialog)
- NOTIFY (outside of a dialog)
- OPTIONS (outside of a dialog)

Applications configure authentication quadruplets for virtual board by constructing a GC_PARM_BLK that contains a separate parameter element for each quadruplet, then calling the **gc_SetAuthenticationInfo( )** function with that parameter block. Authentication quadruplets are removed in the same way but using a different parameter ID in the parameter element. The same function call can configure or remove any number of quadruplets for a given virtual board by including the appropriate combination of parameter elements in the GC_PARM_BLK. For a given function call, each parameter in the GC_PARM_BLK should have a unique realm/identity pair; if multiple parameter elements have the same realm/identity pair, only the the last of these elements in the parameter block becomes effective.

To add or modify an authentication quadruplet, the relevant set ID and parameter ID are:

IPSET_CONFIG
IPPARM_AUTHENTICATION_CONFIGURE

- parameter value is an IP_AUTHENTICATION data structure containing the desired quadruplet values. If the realm/identity pair is unique for the virtual board, a new quadruplet is added to the library's authentication database. If the realm/identity pair matches an existing quadruplet, the existing username/password pair is replaced by the new username/password pair.

To remove an existing authentication quadruplet, the relevant set ID and parameter ID are:

IPSET_CONFIG

IPPARM_AUTHENTICATION_REMOVE

- parameter value is an IP_AUTHENTICATION data structure that identifies the realm and identity of the quadruplet to be removed. The username and password elements of this structure are ignored. If the specified realm and identity do not match those of an existing quadruplet, the function call produces an IPERR_UNAVAILABLE error.

The elements of the authentication quadruplets are contained in an IP_AUTHENTICATION data structure, with each element having the following characteristics:

realm

a case-insensitive string that defines the protected domain name. This element must always contain a non-empty string.

identity

for a single-user realm, an empty string

for a multi-user realm, either a case-insensitive string that identifies the user in the given realm, or else an empty string to allow specification of a default username/password pair. Non-empty strings must conform to the conventions for a SIP URI, and must begin with a "sip:" or "sips:" scheme

username

a case-sensitive, null-terminated string that is the user's name. This element must always contain a non-empty string when configuring an authentication quadruplet. This value of this structure element is ignored when removing an authentication quadruplet.

password

a case-sensitive, null-terminated string that is the user's secret password in clear text. This element can optionally be an empty string, for example, if the quadruplet contains an anonymous username. This value of this structure element is ignored when removing an authentication quadruplet.

When preparing to configure a quadruplet, the application should begin by initializing the IP_AUTHORIZATION structure with the **INIT_IP_AUTHORIZATION( )** function, which configures the structure with the correct version number and with NULL string pointers for each element. The application should then populate each element with the desired string, including any empty strings. If any of the elements is left with a NULL pointer when passed to the function, the function call fails with IPERR_BAD_PARM.

Note that the **gc_SetConfigData( )** and **gc_SetUserInfo( )** functions **cannot** be used to configure authentication quadruplets. If a GC_PARM_BLK containing either of the authentication parameter IDs is passed to either of those functions, the function call fails with IPERR_BAD_PARM.

4.25 Call Transfer

The Global Call library provides six APIs specifically for call transfer in the IP technology. These APIs are described in the *Global Call API Library Reference* with protocol-specific variances described in the subsections of Section 7.3, "Global Call Function Variances for IP". This section describes general considerations for implementing call transfer as well as details specific to H.450.2 (part of the H.323 protocol suite) and SIP protocols. For H.450.2-specific call transfer scenarios see Section 3.2, "Call Transfer Scenarios When Using H.323", on page 50, and for SIP-specific call transfer scenarios, see Section 3.3, "Call Transfer Scenarios When Using SIP", on page 67. The topics covered here include:

- Enabling Call Transfer
- Global Call Line Devices for Call Transfer
- Incoming Transferred Call
- Call Transfer Glare Condition
- Call Transfer When Using SIP

4.25.1 Enabling Call Transfer

The call transfer supplementary service is a feature that must be enabled at the time the **gc_Start( )** function is called. Both H.450.2 and SIP call transfer services are enabled at the same time.

The **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions, which must be called before the **gc_Start( )** function, populate the IPCCLIB_START_DATA and IP_VIRTBOARD structures, respectively, with default values. The default value of the sup_serv_mask field in the IP_VIRTBOARD structure disables the call transfer service for both H.323 and SIP protocols. The default sup_serv_mask field value must therefore be overridden with the value IP_SUP_SERV_CALL_XFER for each IPT board device on which call transfer is to be enabled. The following code snippet provides an example for two virtual boards:

```
.
.
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
.
.
```

Note: If the application tries to use one of the six IP call transfer functions when call transfer was not explicitly enabled via the IP_VIRTBOARD structure during **gc_Start( )**, the function call fails with an IPERR_SUP_SERV_DISABLED indication.

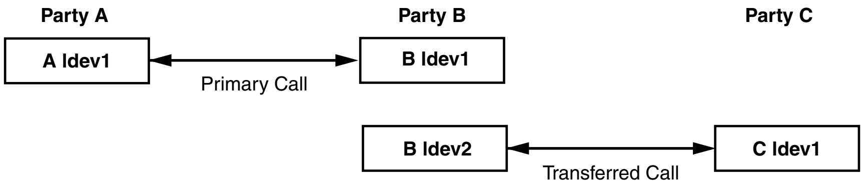
4.25.2 Global Call Line Devices for Call Transfer

The Global Call IP architecture is designed so that each RTP transcoder at all times is streaming (xmit and rcv) with only one other endpoint. In order to support call transfers, two Global Call line devices are required at some or all of the endpoints. And because all involved call handles must be on the same stack instance, the following limitations are imposed on call transfers:

- When performing an attended call transfer at party A, both the consultation line device and the transferring line device must be on the same virtual board.
- When performing a call transfer (either attended or unattended) at party B, both the transferring line device and the transferred line device must be on the same virtual board.
- When performing an attended call transfer at party C, both the consultation line device and the transferred-to line device must be on the same virtual board.

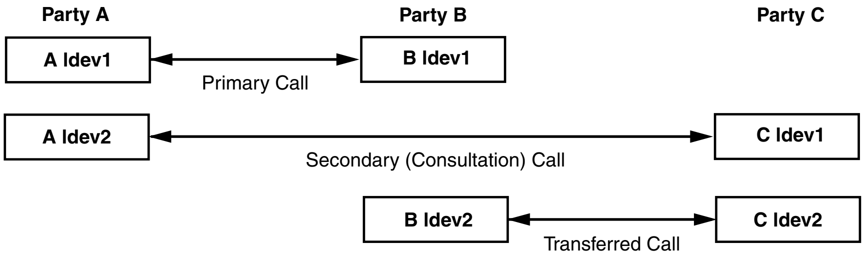
To support blind call transfer, two Global Call line devices are required at the transferred (party B) endpoint, one for the primary call with the transferring (party A) endpoint and a second to initiate the transferred call to the transferred-to (party C) endpoint. See Figure 44.

Figure 44. Global Call Devices for H.450.2 Blind Call Transfer or SIP Unattended Transfer



To support a successful H.450.2 supervised call transfer or SIP attended call transfer, two Global Call line devices are eventually utilized at all endpoints. The transferring endpoint or transferor (party A) makes a consultation call to the transferred-to endpoint or transfer target (party C), thus utilizing two line devices at both these endpoints as well. See Figure 45.

Figure 45. Global Call Devices for Supervised Call Transfer



4.25.3 Incoming Transferred Call

The incoming transferred call to party C contains the call control library (CCLIB) cause value of IPEC_IncomingTransfer and a Global Call library (GC LIB) cause value of GCRV_XFERCALL. The **gc_ResultInfo( )** function can be used to retrieve these values.

In the case of supervised transfer, the associated CRN of the secondary/consultation call is provided. The secondary CRN can be accessed via the extevtdatap pointer within the METAEVENT structure of the GCEV_OFFERED event which references a GC_PARM_BLK. From this parameter block, a data element identified by the SetId/ParmId pair of GCSET_SUPP_XFER and GCPARM_SECONDARYCALL_CRN can be retrieved via the parameter block utility functions to retrieve the secondary call CRN, which is of datatype size CRN (long).

If the transferee address is also provided to party C (optional for H.450.2), it can also be retrieved from this same parameter block, via a data element identified by the SetId/ParmId pair of GCSET_SUPP_XFER and GCPARM_TRANSFERRING_ADDR via the parameter block utility functions as a character array of maximum size GC_ADDRSIZE.

The following code sample demonstrates how to implement this:

```
   .
   .
   .
case GCEV_OFFERED:
{
   if (metaevent.extevtdatap)
   {
      GC_PARM_BLKP parm_blkp = metaevent.extevtdatap;
      GC_PARM_DATAP curParm = NULL;
      printf("GCEV_OFFERED has parmblk:\n");
      while ((curParm = gc_util_next_parm(parm_blkp, curParm)) != NULL)
      {
         CRN secondaryCRN = 0;
         char transferringAddr[GC_ADDRSIZE];
         printf("SetID: 0x%x  ParmID: 0x%x\n",curParm->set_ID,curParm->parm_ID);

         switch (curParm->parm_ID)
         {
            case GCPARM_SECONDARYCALL_CRN:
               memcpy(&secondaryCRN, curParm->value_buf, curParm->value_size);
               printf("GCPARM_SECONDARYCALL_CRN: 0x%x\n",secondaryCRN);
               break;

            case GCPARM_TRANSFERRING_ADDR:
               memcpy(transferringAddr, curParm->value_buf, curParm->value_size);
               printf("GCPARM_TRANSFERRING_ADDR: %s\n",transferringAddr);
               break;

            default:
               printf("UNEXPECTED PARM_ID: %d\n",curParm->parm_ID);
               break;
         }
      }
   }
break;
.
.
.
```

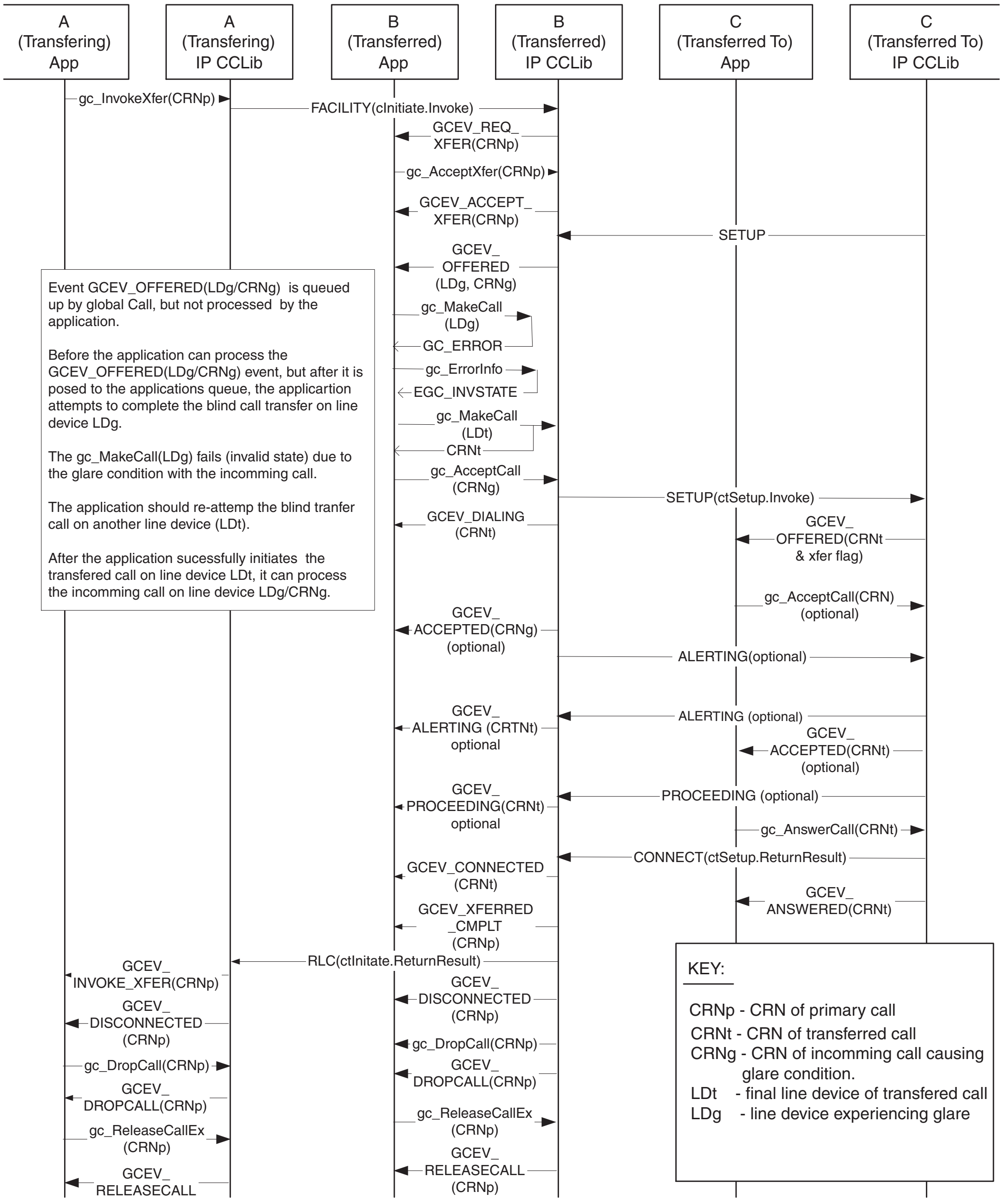
4.25.4 Call Transfer Glare Condition

Glare can occur on a line device during both blind and supervised call transfer operations. Glare occurs on a line device during call transfer at Party B when the application calls **gc_MakeCall( )** to establish the transferred call (after the application has called **gc_AcceptXfer( )** on the primary

CRN). Glare occurs because the CCLIB IP library has chosen the same line device for an incoming call that the application has chosen for establishing the transferred call. The application indication that this glare condition has occurred is that **gc_MakeCall( )** fails with an error indication of EGC_INVSTATE, GCRV_GLARE, or EGC_ILLSTATE. The application should retry the transferred call establishment request on another "available" line device. The application should process the GCEV_OFFERED metaevent on the incoming call/line device that caused the glare "normally" when it is retrieved. The call scenario in Figure 46 describes the glare condition and the appropriate application response.

Figure 46. Call Transfer Glare Condition

Precondition: Primary call between A and B is connected (not shown).



Event GCEV_OFFERED(LDg/CRNg) is queued up by global Call, but not processed by the application.

Before the application can process the GCEV_OFFERED(LDg/CRNg) event, but after it is posed to the applications queue, the applicartion attempts to complete the blind call transfer on line device LDg.

The gc_MakeCall(LDg) fails (invalid state) due to the glare condition with the incomming call.

The application should re-attemp the blind tranfer call on another line device (LDt).

After the application sucessfully initiates the transfered call on line device LDt, it can process the incomming call on line device LDg/CRNg.

KEY:

CRNp - CRN of primary call
CRNt - CRN of transferred call
CRNg - CRN of incomming call causing glare condition.
LDt - final line device of transfered call
LDg - line device experiencing glare

Post Condition: Transferred call between B and C Completed. Primary call between A and B is dropped and released. Incoming call that causes glare is ringing.

4.25.5 Call Transfer When Using SIP

This section describes specific call transfer procedures when using SIP protocol. For complete SIP-specific call transfer scenarios see Section 3.3, “Call Transfer Scenarios When Using SIP”, on page 67. The topics covered here include:

- Enabling GCEV_INVOKE_XFER_ACCEPTED Events
- Invoking an Unattended Call Transfer
- Invoking an Attended Call Transfer
- Processing Asynchronous Call Transfer Events
- Handling a Transfer Request
- Making a Transferred Call

4.25.5.1 Enabling GCEV_INVOKE_XFER_ACCEPTED Events

The following code snippet illustrates how to enable the GCEV_INVOKE_XFER_ACCEPTED event type, which is optionally used to notify the application at party A that party B has accepted a transfer request. This event type is disabled by default. This event can be enabled for an individual line device at any time after the line device is opened. The event is enabled in the party A (Transferor) application, and need only be enabled if the application wishes to receive the events. Note that there is no equivalent event in H.450.2.

```
//enable GCEV_INVOKE_XFER_ACCEPTED event

GC_PARM_BLK *t_pParmBlk = NULL;
long request_id;

gc_util_insert_parm_val(&t_pParmBlk, GCSET_CALLEVENT_MSK, GCACT_ADDMSK,
                        sizeof(long), GCMSK_INVOKEXFER_ACCEPTED);

gc_SetConfigData(GCTGT_GCLIB_CHAN,ldev,t_pParmBlk, 0, GCUPDATE_IMMEDIATE, &request_id, EV_SYNC);

gc_util_delete_parm_blk(t_pParmBlk);
```

Disabling the event is done in exactly the same way except that the parameter ID that is set in the GC_PARM_BLK would be GCACT_SUBMSK instead of GCACT_ADDMSK.

4.25.5.2 Invoking an Unattended Call Transfer

The following code snippet illustrates how to invoke an unattended (blind) transfer on a channel that is in the connected state. In this example, the Refer-To header field of the REFER message that is sent is set to “sip:500@192.168.1.10”, while the Referred-By header field is automatically populated by Global Call.

```
int Gc_InvokeXfer(int channel)
{
   INT32  rc;
   GCLIB_MAKECALL_BLK t_gclibmakecallblk;
   GC_MAKECALL_BLK    t_gcmakecallblk = {0};
   char invokeaddr[] = "192.168.1.10";  // party C (TRTSE)
   char phonelist[] = "500";
   /* Invoke transfer */
   memset(&t_gclibmakecallblk, 0, sizeof(GCLIB_MAKECALL_BLK));
   strcpy(t_gclibmakecallblk.destination.address, invokeaddr);
   t_gclibmakecallblk.destination.address_type = GCADDRTYPE_IP;
   t_gclibmakecallblk.destination.address_plan = GCADDRPLAN_UNKNOWN;
   t_gcmakecallblk.gclib = &t_gclibmakecallblk;

   gc_util_insert_parm_ref(&t_pParmBlk, IPSET_CALLINFO, IPPARM_PHONELIST,
                           sizeof(phonelist), phonelist);

   t_gclibmakecallblk.ext_datap = t_pParmBlk;

   rc = gc_InvokeXfer(session[channel].crn, 0, 0, &t_gcmakecallblk, 0, EV_ASYNC);

   gc_util_delete_parm_blk(t_pParmBlk);

   if(GC_SUCCESS != rc)
   {
      printf("GC_APP : [%d] Invoke Xfer failed!!!\n",channel);
      return GC_ERROR;
   }

  return GC_SUCCESS;
}
```

4.25.5.3 Invoking an Attended Call Transfer

Note that it is necessary for the consultation call to be in the connected state at **both** parties before the transfer operation is invoked. If the transferred-to party (party C) is a Global Call application and is not in the connected state when the transfer is invoked, it may fail to receive the Global Call event for the transfer request, which will cause a GCEV_TASKFAIL.

The following code snippet illustrates how a party that is connected to two remote parties, a primary call and a secondary call, invokes a call transfer by sending a REFER to one of the remote parties. The Refer-To, Replaces, and Referred-By header fields in the REFER are automatically filled in by Global Call. Note that the application does not have to specify the Refer-To information in an attended transfer because the secondary call already contains that information.

```
int Gc_InvokeXfer(int primaryChannel, int secondaryChannel)
{
   INT32  rc;

   /* Invoke transfer */
   rc = gc_InvokeXfer(session[primaryChannel].crn, session[secondaryChannel].crn,
                      0, 0, 0, EV_ASYNC);

   if(GC_SUCCESS != rc)
   {
      printf("GC_APP : [%d] Invoke Xfer failed!!!\n",primaryChannel);
      return GC_ERROR;
   }

   return GC_SUCCESS;
}
```

4.25.5.4 Processing Asynchronous Call Transfer Events

The following code snippets illustrate how to handle the asynchronous events that notify applications of the call transfer status as a SIP call transfer proceeds.

```
INT32 processEvtHandler()
{
   METAEVENT    metaEvent;
   GC_PARM_BLK  *parmblkp = NULL;
   :

   int  rc = gc_GetMetaEvent(&metaEvent);
   if (GC_SUCCESS != rc)
   {
      printf("GC_APP : gc_GetMetaEvent() failed\n");
      return rc;
   }

   long evtType = sr_getevttype();
   long evtDev = sr_getevtdev();
   int  g_extIndex = g_lArray[g_evtdev];

   switch (evtType)
   {

      *///////////////////////////////////////////
      // Party A events
      ///////////////////////////////////////////

      case GCEV_INVOKE_XFER_ACCEPTED:
         // remote party has accepted REFER by 2xx response
         printf("Invoke Transfer Accepted By Remote\n");
         break;

      case GCEV_INVOKE_XFER:
         // remote party has notified transfer success in NOTIFY
         printf("Invoke Transfer Successful\n");
         break;

      case GCEV_INVOKE_XFER_FAIL:
         // Invoke Transfer failed by remote NOTIFY or locally
         PrintEventError(&metaEvent);
         break;

      case GCEV_INVOKE_XFER_REJ:
         // Invoke Transfer Rejected by Remote party
         PrintEventError(&metaEvent);
         break;

      /////////////////////////////////////////
      // Party B events
      /////////////////////////////////////////

      case GCEV_REQ_XFER:
         // Incoming transfer request
         GC_REROUTING_INFO *pRerouteInfo = (GC_REROUTING_INFO *)metaEvent.extevtdatap;
         printf("Reroute number = %s\n", pRerouteInfo->rerouting_num);

         if(NULL != pRerouteInfo->parm_blkp)
         {
            // Handle parm blocks
         }

         strcpy(session[g_extIndex].rerouting_num,pRerouteInfo->rerouting_num);
         session[g_extIndex].rerouting_addrblk = *pRerouteInfo->rerouting_addrblkp;

         GC_HandleXferReq(g_extIndex)
         break;

      case GCEV_ACCEPT_XFER:
         // Accepted incoming transfer request
         break;
      case GCEV_ACCEPT_XFER_FAIL:
         // Failed to accept incoming transfer request
         PrintEventError(&metaEvent);
         break;

      case GCEV_REJ_XFER:
         // Rejected incoming transfer request
         break;

      case GCEV_REJ_XFER_FAIL:
         // Failed to reject incoming transfer request
         PrintEventError(&metaEvent);
         break;

      case GCEV_XFER_CMPLT:
         // completed transferred call
         break;

      case GCEV_XFER_FAIL:
         // Failed to complete the transferred call
         PrintEventError(&metaEvent);
         break;

      /////////////////////////////////////////
      // Party C events
      /////////////////////////////////////////

      case GCEV_OFFERED:
         // Received incoming call
         // Normall incoming call handling
         ...
         break;

      ...
   }
   ...
}



void PrintEventError(METAEVENT* pEvent, long evtDev)
{
   int gcError;    /* GlobalCall Error */
   int ccLibId;    /* CC Library ID */
   long ccError;   /* Call Control Library error code */
   char *GCerrMsg; /* GC pointer to error message string */
   char *errMsg;   /* CCLIB pointer to error message string */

   if(gc_ResultValue(pEvent, &gcError, &ccLibId, &ccError)   == GC_SUCCESS)
   {
      gc_ResultMsg(LIBID_GC, (long) gcError, &GCerrMsg);
      gc_ResultMsg(ccLibId, ccError, &errMsg);

      printf("Ld 0x%lx, GC (%d) %s, CC (%ld) %s, (%s)\n",
             evtDev, gcError, GCerrMsg, ccError, errMsg, ATDV_NAMEP(evtDev));
   }
}
```

4.25.5.5 Handling a Transfer Request

The following code snippet illustrates how party B handles an incoming transfer request (REFER). Party B can either reject the request or accept it. Note that if no rejection reason is specified, the default reason, 603 Decline, is used.

```
int Gc_HandleXferReq(int channel)
{
   if(session[channel].ConfigFileParm.autoRejectCallXfer)
   {
      printf("GC_APP : [%d] Reject call xfer request\n",channel);
      if(GC_SUCCESS != gc_RejectXfer(session[channel].crn, IPEC_SIPReasonStatus502BadGateway,
                                    0, EV_ASYNC))
      {
         printf("GC_APP : [%d] Reject call xfer failed on device 0x%lx\n", channel,
                session[channel].ldev);
         PrintEventError(g_evtdev);
         return GC_ERROR;
      }
   }
   else
   {
      printf("GC_APP : [%d] Accept call xfer request\n",channel);
      if(GC_SUCCESS != gc_AcceptXfer(session[channel].crn, 0, EV_ASYNC))
      {
         printf("GC_APP : [%d] Accept call xfer failed on device 0x%lx\n", channel,
                 session[channel].ldev);
         PrintEventError(g_evtdev);
         return GC_ERROR;
      }
   }

   return GC_SUCCESS;
}
```

4.25.5.6 Making a Transferred Call

The following code snippet illustrates how party B makes the transferred call to party C after accepting transfer request from party A

```
int Gc_MakeXferCall(int channelPrimary, int channelXfer)
{

   GC_PARM_BLK         * t_pParmBlk = NULL;
   GCLIB_MAKECALL_BLK  t_gclibmakecallblk ;
   GC_MAKECALL_BLK     t_gcmakecallblk = {0};
   t_gcmakecallblk.gclib = &t_gclibmakecallblk;
   int                 channelXfer;

   memset(&t_gclibmakecallblk, 0, sizeof(GCLIB_MAKECALL_BLK));

   gc_util_insert_parm_val(&t_pParmBlk, GCSET_SUPP_XFER, GCPARM_PRIMARYCALL_CRN,
                           sizeof(unsigned long), session[channelPrimary].crn);

   t_gclibmakecallblk.ext_datap = t_pParmBlk;
   t_gclibmakecallblk.destination = session[channelPrimary].rerouting_addrblk;

   int frc = gc_MakeCall(session[channelXfer].ldev, &session[channelXfer].crn,
                         NULL, &t_gcmakecallblk, 0, EV_ASYNC);

   if((GC_SUCCESS != frc) ||(0 == session[channelXfer].crn))
   {
      printf("GC_APP : [%d] Gc_MakeCall failed: : crn 0x%lx\n", channelXfer,
              session[channelXfer].crn);
      PrintGCError(session[channelXfer].ldev);
   }

   gc_util_delete_parm_blk(t_pParmBlk);

   return GC_SUCCESS;
}
```

4.26 T.38 Fax Server

Global Call support for T.38 Fax Server is described under the following topics:

- T.38 Fax Server Support Overview
- Specifying Manual Operating Mode
- Initiating a Switch from Audio to T.38 Fax
- Associating a T.38 Fax Device with a Media Device When a Fax Request is Received
- Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax
- Sending a T.38 Fax in a Session Without Audio Established
- Receiving a T.38 Fax in a Session Without Audio Established
- Sending a Request to Switch from T.38 Fax to Audio
- Receiving a Request to Switch from T.38 Fax to Audio

4.26.1 T.38 Fax Server Support Overview

Global Call provides T.38 fax server functionality to support fax-on-demand and other applications. The functionality includes the ability of an application to:

- initiate and complete a T.38 session without an audio connection being first established
- switch coders from audio to T.38 fax and back again during a pre-established audio connection

To support T.38 fax functionality, Global Call uses two types of media devices:

- a traditional Media device
- a new T.38 Fax device

By default, ipmBxCy represents the media device on board x and channel y, which has no fax capability on HMP. By associating the corresponding voice handle with a fax handle, the ipmBxCy device represents the fax channel defined by the fax handle, with no voice capability. Disassociating the voice and fax devices restores the ipmBxCy device voice capability.

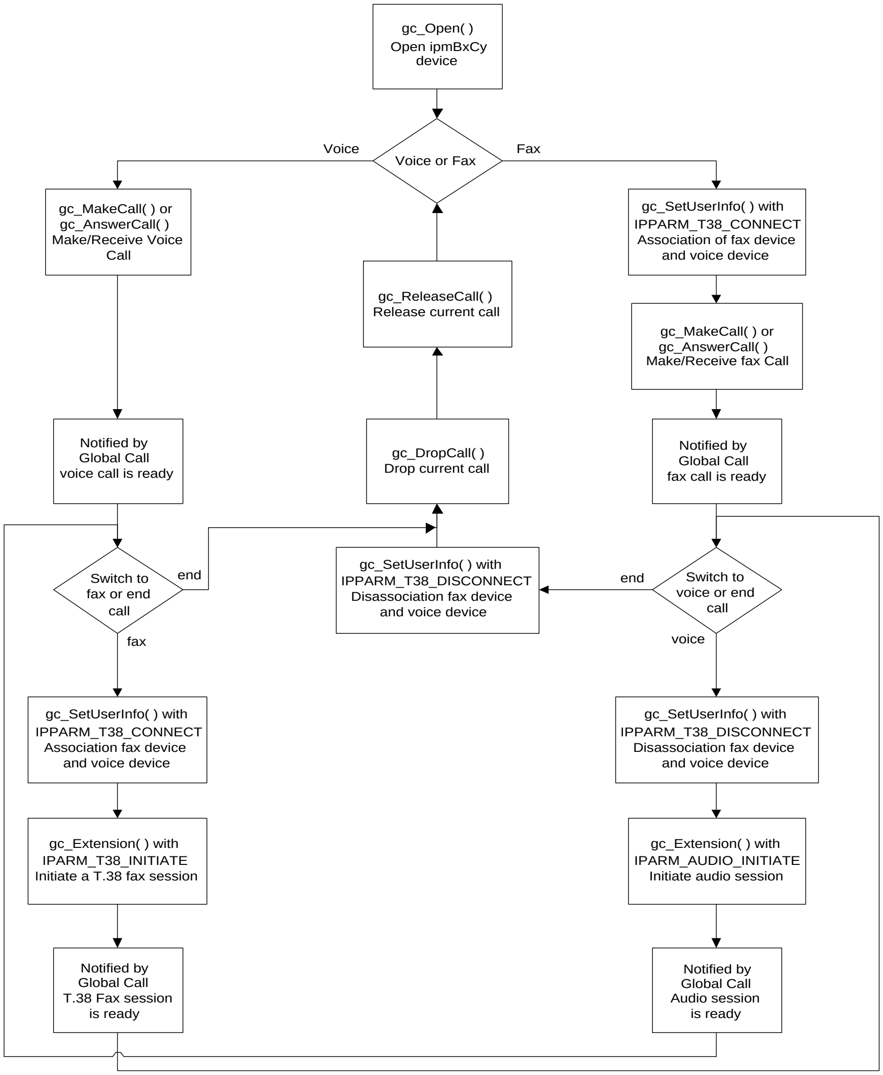
Global Call uses the **gc_SetUserInfo( )** function to associate and disassociate a traditional Media device with a T.38 Fax device when establishing or concluding a T.38 fax connection. Manual device association must be done before the next Global Call function that requires fax information:

- For H.323, the association must be made before **gc_MakeCall( )** on the outbound call side, and **gc_CallAck( )**, **gc_AcceptCall( )** and **gc_AnswerCall( )** on the inbound call side, whichever occurs first since the underlying “open logical channel” can happen at any of these times if coder capabilities and fax port information is available.
- For SIP, the association must be made before **gc_MakeCall( )** on the outbound call side and **gc_AnswerCall( )** on the inbound call side, since media can only be opened after either of these functions.

Note: When a Media device is associated with a T.38 Fax device to establish a fax session over an existing audio connection, then when the fax session concludes, the Media device must be disassociated with the T.38 Fax device, **optionally** reestablishing the audio connection, **before** the call is dropped.

Figure 47 provides a flowchart that summarizes the T.38 fax server functionality and indicates the Global Call functions and events used at different stages in the call control process. The initial voice or fax capability decision before call connection is determined as described in Section 4.3.2.1, “Specifying Media Capabilities Before Connection”, on page 110.

Figure 47. T.38 Fax Server Support in Manual Mode


gc_Open( )
Open ipmBxCy device
Voice or Fax
Voice
Fax
gc_MakeCall( ) or gc_AnswerCall( ) Make/Receive Voice Call
gc_SetUserInfo( ) with IPPARM_T38_CONNECT Association of fax device and voice device
gc_ReleaseCall( ) Release current call
gc_MakeCall( ) or gc_AnswerCall( ) Make/Receive fax Call
Notified by Global Call voice call is ready
gc_DropCall( ) Drop current call
Notified by Global Call fax call is ready
Switch to fax or end call
end
fax
gc_SetUserInfo( ) with IPPARM_T38_DISCONNECT Disassociation fax device and voice device
end
Switch to voice or end call
voice
gc_SetUserInfo( ) with IPPARM_T38_CONNECT Association fax device and voice device
gc_SetUserInfo( ) with IPPARM_T38_DISCONNECT Disassociation fax device and voice device
gc_Extension( ) with IPARM_T38_INITIATE Initiate a T.38 fax session
gc_Extension( ) with IPARM_AUDIO_INITIATE Initiate audio session
Notified by Global Call T.38 Fax session is ready
Notified by Global Call Audio session is ready

4.26.2 Specifying Manual Operating Mode

An application must be configured in “Manual” mode to control the association and disassociation of Media and T.38 Fax devices during each call. The mode of operation is set on a board device basis. Once the GCEV_OPENEX event is received to confirm that the board device is open, the **gc_SetConfigData( )** function can be used to configure “Manual” mode as indicated in the code example below:

```
INT32 processEvtHandler()
{
   GC_PARM_BLK  *parmblkp = NULL;
   long         t = 0;
   :
   :
   switch (evtType)
   {
      :
      :
      case GCEV_OPENEX:
          gc_util_insert_parm_val(&parmblkp, IPSET_CONFIG, IPPARM_OPERATING_MODE,
                                  sizeof(int), IP_MANUAL_MODE);

          gc_SetConfigData(GCTGT_CCLIB_NETIF, pline->ldev, parmblkp, 0,
                           GCUPDATE_IMMEDIATE, &t, EV_ASYNC);

          gc_util_delete_parm_blk(parmblkp);
       break;
      :
      :
   }
   :
   :
}
```

4.26.3 Initiating a Switch from Audio to T.38 Fax

After an audio session has been established, the application can use the **gc_Extension( )** function to initiate a RequestMode (H.323) or Reinvite (SIP) to change the coder. Prior to initiating a coder change, the T.38 Fax device must be associated with the Media device. This can be achieved using the **gc_SetUserInfo( )** function. The application receives a GCEV_EXTENSION event to indicate that T.38 media is ready to send and receive fax information. The following code provides an example:

```
INT32 processEvtHandler()
{
   METAEVENT      metaEvent;
   GC_PARM_BLK    *parmblkp = NULL;
   IP_CONNECT     ipConnect;
     :
   switch (evtType)
   {
     :
      case GCEV_CONNECTED:
         /* received Connect event */
         /* in conversation */
         ipConnect.version = 0x100;
         ipConnect.mediaHandle = pline->mediaH;
         ipConnect.faxHandle = pline->faxH;
         ipConnect.connectType = IP_FULLDUP;

         gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_CONNECT,
                                 (sizeof(IP_CONNECT)), (void *)(&ipConnect));
         gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
         gc_util_delete_parm_blk(parmblkp);
         /* Initiate T.38 codec switch */
         gc_util_insert_parm_ref(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_T38_INITIATE,
                                 sizeof(int), NULL);
         gc_Extension(GCTGT_GCLIB_CRN,pline->crn, IPEXTID_CHANGEMODE,
                      parmblkp, NULL, EV_ASYNC);
         gc_util_delete_parm_blk(parmblkp);
      break;

      case GCEV_EXTENSIONCMPLT:
             /* received extension complete event for T.38 initiation*/
             /* do nothing */
      break;

      case GCEV_EXTENSION:
             /* received extension event for media readiness */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;

         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_READY:
                        /* Ready to send and receive fax */
                        fx_sendfax();
                     break;
                     :
                     :
                  }
               break;
            }
         }

      break;
      :
   }
   :
}
```

4.26.4 Associating a T.38 Fax Device with a Media Device When a Fax Request is Received

During a voice call, a T.38 Fax request can be received by a RequestMode (H.323) or Reinvite (SIP) message. The application receives notification of the request as a GCEV_EXTENSION event. A T.38 Fax device must then be associated with the Media device by filling in an IP_CONNECT structure with the appropriate T.38 Fax and Media device handles and using the **gc_SetUserInfo( )** function. To continue to accept the request, the **gc_Extension( )** function is used as described in Section 4.26.5, "Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax", on page 251. The following code provides an example:

```
INT32 processEvtHandler()
{
   METAEVENT        metaEvent;
   GC_PARM_BLK      *parmblkp = NULL;
   GC_PARM_DATAP    t_gcParmDatap = NULL;
   GC_PARM_BLK      *parmblkp2 = NULL;
   EXTENSIONEVTBLK  *ext_evtblkp = NULL;
   IP_CONNECT       ipConnect;
   :
```

```
   switch (evtType)
   {
      case GCEV_EXTENSION:
         /* received extension event, parse PARM_BLK examine
          * extension data
          */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;
         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_T38_REQUESTED:
                         /* connect the media and fax devices */
                         ipConnect.version = 0x100;
                         ipConnect.mediaHandle = pline->mediaH;
                         ipConnect.faxHandle = pline->faxH;
                         ipConnect.connectType = IP_FULLDUP;

                         gc_util_insert_parm_ref(&parmblkp2, IPSET_FOIP, IPPARM_T38_CONNECT,
                                                 sizeof(IP_CONNECT), (void *)(&ipConnect));
                         gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn,
                                        parmblkp, GC_SINGLECALL);
                         gc_util_delete_parm_blk(parmblkp2);

                         /* accept T.38 request by example 4.17.5 */
                         acceptCodecSwitchRequest();
                      break;

                     case IPPARM_READY:
                        /* Ready to send and receive fax */
                        fx_sendfax();
                     break;
                  }
               break;
            }
         }
      break;
   }
   :
}
```

4.26.5 Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax

After T.38 coder change request has been received, followed by association of T.38 Fax device with Media device as described in Section 4.26.4, "Associating a T.38 Fax Device with a Media Device When a Fax Request is Received", on page 250, the application can use the **gc_Extension( )** function to accept or reject the request as follows:

- To accept the request, the GCPARM_BLK associated with the **gc_Extension( )** function includes components that indicate acceptance, specifically IPSET_SWITCH_CODEC and IPPARM_ACCEPT. A RequestModeAck (H.323) or 200 OK (SIP) message is not sent until the request is accepted. The following code provides an example:

```
/* Reject the incoming request */

INT32 acceptCodecSwitchRequest()
{
   GC_PARM_BLK *parmblkp = NULL;
   :
   gc_util_insert_parm_val(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_ACCEPT,
                           sizeof(int), NULL);
   gc_Extension(GCTGT_GCLIB_CRN,pline->crn, PEXTID_CHANGEMODE,
                parmblkp, NULL, EV_ASYNC);
   gc_util_delete_parm_blk(parmblkp);

}
```

- To reject the request, the GCPARM_BLK associated with the **gc_Extension( )** function includes components that indicate rejection, specifically IPSET_SWITCH_CODEC and IPPARM_REJECT. The reason for rejecting the request is also included in the GCPARM_BLK. Chapter 10, “IP-Specific Event Cause Codes” describes the supported reject reasons that can be used in this context. For H.323, reasons prefixed by “IPEC_Q931Cause” can be used. For SIP, reasons prefixed by “IPEC_SIPReason” can be used. The reason parameter corresponds to a RequestModeReject cause (H.323) or a negative response code (SIP). The following code provides an example:

```
/* Reject the incoming request */

INT32 rejectCodecSwitchRequest()
{
   GC_PARM_BLK *parmblkp = NULL;
   :
   :
   /* Reject with reason being busy, SIP */
   gc_util_insert_parm_val(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_REJECT,
                           sizeof(int), IPEC_SIPReasonStatus486BusyHere);
   gc_Extension(GCTGT_GCLIB_CRN, pline->crn, IPEXTID_CHANGEMODE,
                parmblkp, NULL, EV_ASYNC);
   gc_util_delete_parm_blk(parmblkp);
}
```

4.26.6 Sending a T.38 Fax in a Session Without Audio Established

Global Call supports the transmission of fax information in a session that does not already have an audio connection established. To send T.38 Fax in such a session, the application must use the **gc_SetConfigData( )** function to specify “Manual” mode, then associate a T.38 Fax device with a media device before calling the **gc_MakeCall( )** function to actually send the fax information. The association only applies to a single call and can be accomplished by calling the **gc_SetUserInfo( )** function on a line device for a single call, or in the GC_MAKECALL_BLK structure when calling **gc_MakeCall( )**.

Note: If using **gc_SetUserInfo( )** to make the association on a line device, the duration must be set to GC_SINGLECALL rather than GC_ALLCALLS.

The following code provides an example:

```
INT32 processEvtHandler()
{
   GC_PARM_BLK *parmblkp = NULL;
   :
   :
   switch (evtType)
   {
      case GCEV_OPENEX:
         /* Set manual mode */
         gc_util_insert_parm_val(&parmblkp, IPSET_CONFIG, IPPARM_OPERATING_MODE,
               sizeof(int), IP_MANUAL_MODE);
         gc_SetConfigData(GCTGT_GCLIB_NETIF, boarddev, parmblkp, 0,
               GCUPDATE_IMMEDIATE, &t, EV_ASYNC);
         gc_util_delete_parm_blk(parmblkp);

         /* Associate T.38 device with media device */
         ipConnect.version = 0x100;
         ipConnect.mediaHandle = pline->mediaH;
         ipConnect.faxHandle = pline->faxH;
         ipConnect.connectType = IP_FULLDUP;
         gc_util_insert_parm_ref(&(libBlock.ext_datap), IPSET_FOIP, IPPARM_T38_CONNECT,
              sizeof(IP_CONNECT), (void *)(&ipConnect));
         gc_SetUserInfo(GCTGT_GCLIB_CHAN, pline->LDEV, parmblkp, GC_SINGLECALL);
         gc_util_delete_parm_blk(parmblkp);

         /* Make call now */
         gc_MakeCall();
      break;

      case GCEV_CONNECTED:
         fx_sendfax();
      break;
   }
   :
   :
}
```

4.26.7 Receiving a T.38 Fax in a Session Without Audio Established

Global Call supports the reception of fax information in a session that does not already have an audio connection established. The application can receive a GCEV_OFFERED event with a T.38 Fax request even if the session has no audio connection.

Note: The parameter block associated with the GCEV_OFFERED event indicates an incoming T.38 Fax request if T.38 Fax is the **only** media offered in the incoming request. If more than T.38 media is offered, no specific T.38 information will be associated with offered event.

To answer the T.38 offer, the application must associate a Fax device with the Media device and set local T.38 media capability before calling the **gc_AnswerCall( )** function. The following code provides an example:

```
INT32 processEvtHandler()
{
   METAEVENT        metaEvent;
   GC_PARM_BLK      *parmblkp = NULL;
   GC_PARM_DATAP    t_gcParmDatap = NULL;
   GC_PARM_BLK      *parmblkp2 = NULL;
   EXTENSIONEVTBLK  *ext_evtblkp = NULL;
   IP_CONNECT       ipConnect;
   IP_CAPABILITY    ipcap;
   :
   switch (evtType)
   {
      case GCEV_OFFERED:
         /* parse PARM_BLK examine data */
         parmblkp = (GC_PARM_BLK *)metaEvent.extevtdatap;

         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_FOIP:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_T38_OFFERED:
                         /* connect media with fax devices */
                         ipConnect.version = 0x100;
                         ipConnect.mediaHandle = pline->mediaH;
                         ipConnect.faxHandle = pline->faxH;
                         ipConnect.connectType = IP_FULLDUP;

                         gc_util_insert_parm_ref(&parmblkp2, IPSET_FOIP, IPPARM_T38_CONNECT,
                                                 (sizeof(IP_CONNECT)), (void *)(&ipConnect));
                         gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp2, GC_SINGLECALL);
                         gc_util_delete_parm_blk(parmblkp2);

                         /* set T.38 media capability*/
                         ipcap.capability = GCCAP_DATA_t38UDPFax;
                         ipcap.type = GCCAPTYPE_RDATA;
                         ipcap.direction = IP_CAP_DIR_LCLTXRX;
                         ipcap.extra.data.max_bit_rate = 144;

                         gc_util_insert_parm_ref(&parmblkp2, GCSET_CHAN_CAPABILITY,
                                                 IPPARM_LOCAL_CAPABILITY,
                                                 sizeof(IP_CAPABILITY), &ipcap);
                         gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, pParmBlock2,
                                        GC_SINGLECALL);
                         gc_util_delete_parm_blk(pParmBlock2);

                         /* received completion event for gc_Extension() */
                         gc_AnswerCall(pline->crn, 0, EV_ASYNC);
                      break;
                  }
               break
            }
         }
      break
   }
}
```

4.26.8 Sending a Request to Switch from T.38 Fax to Audio

To request a switch from a T.38 Fax session back to an audio session, the application uses the **gc_Extension( )** function, which initiates a RequestMode (H.323) or Reinvite (SIP) message to actually perform the action. Before initiating the change of coder, the Fax device must be disassociated from the Media device using the **gc_SetUserInfo( )** function. The application receives a GCEV_EXTENSION event to indicate that audio can now be sent and received. The following code provides and example:

```
INT32 switchFromFaxToAudio()
{
   GC_PARM_BLK      *parmblkp = NULL;
   IP_CONNECT       ipConnect;
   ipConnect.version = 0x100;
   ipConnect.mediaHandle = pline->mediaH;

   gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                           (sizeof(IP_CONNECT)), (void *)(&ipConnect));
   gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
   gc_util_delete_parm_blk(parmblkp);

   /* Initiate audio codec switch */
   gc_util_insert_parm_ref(&parmblkp, IPSET_SWITCH_CODEC,
                            IPPARM_AUDIO_INITIATE, sizeof(int), NULL);
   gc_Extension(GCTGT_GCLIB_CRN,pline->crn, IPEXTID_CHANGEMODE, parmblkp, NULL, EV_ASYNC);
   gc_util_delete_parm_blk(parmblkp);
}

INT32 processEvtHandler()
{
   METAEVENT       metaEvent;
   GC_PARM_BLK     *parmblkp = NULL;
   :
   switch (evtType)
   {
      case GCEV_EXTENSIONCMPLT:
         /* received extension complete event for audio initiation*/
         /* do nothing */
      break;

      case GCEV_EXTENSION:
         /* received extension event for media readiness */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;

         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_READY:
                        /* Ready to send and receive audio */
                        gc_Listen();
                     break;
                     :
                     :
                  }
               break;
            }
         }
      break;
     :
   }
 :
}
```

4.26.9 Receiving a Request to Switch from T.38 Fax to Audio

An application may receive a request to switch from a T.38 Fax session back to an audio session. The request is received as a GCEV_EXTENSION event that is triggered by a RequestMode (H.323) or Reinvite (SIP) message. Before accepting the incoming request, the application must disassociate the T.38 Fax device from the Media device using the **gc_SetUserInfo( )** function, then continue accepting the request as described in Section 4.26.5, “Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax”, on page 251.

```
INT32 processEvtHandler()
{
   METAEVENT        metaEvent;
   GC_PARM_BLK      *parmblkp = NULL;
   GC_PARM_DATAP    t_gcParmDatap = NULL;
   GC_PARM_BLK      *parmblkp2 = NULL;
   EXTENSIONEVTBLK  *ext_evtblkp = NULL;
   IP_CONNECT       ipConnect;
   :
   switch (evtType)
   {
      case GCEV_EXTENSION:
         /* received extension event, parse PARM_BLK examine
          * extension data
          */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;
         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_AUDIO_REQUESTED:
                        /* disconnect the media and fax devices */
                        ipConnect.version = 0x100;
                        ipConnect.mediaHandle = pline->mediaH;

                        gc_util_insert_parm_ref(&parmblkp2, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                                                sizeof(IP_CONNECT), (void *)(&ipConnect));
                        gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
                        gc_util_delete_parm_blk(parmblkp2);


                        /* accept audio request by example 4.3.3 */
                        acceptCodecSwitchRequest();
                     break;

                     case IPPARM_READY:
                        /* Ready to send and receive audio */
                        gc_Listen();
                     break;
                  }
               break;
               :
            }
         }
      break;
     :
   }
 :
}
```

4.26.10 Terminating a Call After a T.38 Fax Session

After a T.38 fax session is finished, and prior to issuing **gc_DropCall( )**, the T.38 Fax device needs to be disassociated from the Media device using the **gc_SetUserInfo( )** function. The following code provides and example.

```
INT32 processEvtHandler()
{
METAEVENT       metaEvent;
GC_PARM_BLK     *parmblkp = NULL;
IP_CONNECT      ipConnect;
   :
   switch (evtType)
      {
         case GCEV_DISCONNECTED:
            /* received extension event, parse PARM_BLK examine extension data */

            /* disconnect the media and fax devices */
            ipConnect.version = 0x100;
            ipConnect.mediaHandle = pline->mediaH;

            gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                                    sizeof(IP_CONNECT), (void *)(&ipConnect));
            gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
            gc_util_delete_parm_blk(parmblkp);

            /* dropcall */
            gc_DropCall(pline->crn, GC_NORMAL_CLEARING, EV_ASYNC);
         break;
      }
   :
}
```

4.27 Using Object Identifiers

Object Identifiers (OIDs) are not free strings, they are standardized and assigned by various controlling authorities such as, the International Telecommunications Union (ITU), British Standards Institute (BSI), American National Standards Institute (ANSI), Internet Assigned Numbers Authority (IANA), International Standards Organization (ISO), and public corporations. Depending on the authority, OIDs use different encoding and decoding schemes. Vendors, companies, governments and others may purchase one or more OIDs to use while communicating with another entity on the network. For more information about OIDs, see *http://www.alvestrand.no/objectid/*.

An application may want to convey an OID to the remote side. This can be achieved by setting the OID string in any nonstandard parameter that can be sent in any Setup, Proceeding, Alerting, Connect, Facility, or User Input Indication (UII) message.

Global Call supports the use of any valid OID by allowing the OID string to be included in the GC_PARM_BLK associated with the specific message using the relevant parameter set ID and parameter IDs. Global Call will not send an OID that is not in a valid format. For more information on the valid OID formats see *http://asn-1.com/x660.htm* which defines the general procedures for the operation of OSI (Open System Interconnection) registration authorities.

The application is responsible for the validity and legality of any OID used.

4.28 Host LAN Disconnection Alarms

The Global Gall IP Call Control library allows applications to receive notification of a disruption of traffic over the host NIC. This notification uses the standard GCAMS alarm mechanism.

4.28.1 Signaling LAN Disconnection Alarm

The Global Call IP Call Control library provides facilities to notify applications when there is a disruption of a host LAN connection that is handling call control signaling traffic, and when any such disruption is corrected. The most common cause of such a LAN disruption is cable disconnection, but any disruption of the LAN connection will cause the alarm to be sent to board devices that have registered for it. LAN status is monitored on a 4 second loop.

Signaling LAN disconnect (Alarm State ON) and recovery (Alarm State OFF) alarms are generated on a virtual board device level using the standard GCAMS mechanism. If multiple board devices are connected to different ports on the same NIC (rather than separate NICs), all of those devices that have registered for the alarm will receive alarm events when the NIC's LAN connection fails or when it is restored after a disconnection. There is a single disconnect alarm event and a single corresponding recovery event for each LAN disconnection on each virtual board.

The signaling LAN disconnect and recovery alarms are only reported via asynchronous GCAMS events. There is no mechanism for determining the LAN cable alarm status on demand. The signaling LAN disconnect alarm is not designated as a blocking or non-blocking GCAMS alarm because it is a board device level alarm rather than a line device level alarm. Refer to the *Global Call API Library Reference* and *Global Call API Programming Guide* for more information on GCAMS facilities.

The call control library does not take any action (for example, disconnecting an already set up call) in response to LAN disconnection alarm events. It is up to the application whether or not to take any action when alarm events are received. If the application does not take any action when a LAN disconnect alarm is received, the following behavior applies under the circumstances described:

- Already established calls will not be affected unless the LAN connection that has failed is carrying the media traffic as well as the signaling traffic. (Media LAN disconnection is not reported by the signaling LAN disconnect alarm.)
- A call that is in the process of being established will be disconnected by the Call Control library due to the signaling failure, and the application will be notified of the disconnection via existing Global Call disconnect events with appropriate disconnection reasons.
- If the application ignores the LAN disconnect error and tries to make a new call over the disconnected LAN connection, the call will fail and the application will be notified of the reason via existing Global Call events.

If a LAN disconnection failure occurs during application startup, no GCAMS alarm event will be generated, because there is no virtual board which is started up to receive the alarm. There will also be no alarm events generated for applications using the NIC address associated with the system loopback adapter (typically IP address 127.0.0.1) because the signaling never leaves the system in this case.

To enable the receipt of signaling LAN disconnect alarm events, the application must perform the following general steps:

- Explicitly open the board device.
- Register the device handle (from the open operation) with GCAMS using the Global Call function gc_SetAlarmNotifyAll( ). This registration uses the wildcard Alarm Source Object (ASO) ID, ALARM_SOURCE_ID_NETWORK_ID, because the IP Call Control library ASO ID is not known at this point.

When an alarm event is received, the alarm number, the alarm name, the ASO ID and the ASO name can be retrieved using standard Global Call alarm APIs. The retrieved alarm number is equal to TYPE_LAN_DISCONNECT for a disconnect alarm or TYPE_LAN_DISCONNECT + 0x10 for a reconnect alarm event. The retrieved alarm name will be “Lan Cable Disconnected” or “Lan cable connected”. The retrieved ASO ID will be “IPCCLIBAsoId”.

The following code illustrates how signaling LAN disconnect alarms are enabled and handled.

```
main()
{
   /* Initialize the SRL mode for the application */
   #ifdef _WIN32
      int   mode = SR_STASYNC | SR_POLLMODE;
      /* Set SRL mode */
      sr_setparm(SRL_DEVICE, SR_MODELTYPE, &mode)
   #else
      int   mode = SR_POLLMODE;
      sr_setparm(SRL_DEVICE, SR_MODEID, &mode)
   #endif

   /* Open the board device */
   sprintf(DevName,":N_iptB1:P_IP");
   rc = gc_OpenEx(&boarddev,DevName,EV_ASYNC,(void *)NULL);

   /* Enable Alarm notification on the board handle with generic ASO ID*/
   gc_SetAlarmNotifyAll (boarddev, ALARM_SOURCE_ID_NETWORK_ID, ALARM_NOTIFY);

   /* -- Forever loop where the main work is done - wait for an event or user requested exit */
   for (;;)
   {
      ret = sr_waitevt(500);              /* 1/2 second */
      if (ret != -1)
      {                    /* i.e. not timeout */
         process_event();
      }
   }
}

process_event()
{
   METAEVENT         metaevent;

   gc_GetMetaEvent(&metaevent)
   evttype = metaevent.evttype;

   switch (evttype)
   {
      case GCEV_ALARM:
         print_alarm_info(&metaevent);
         break;
   }
}
```

```
print_alarm_info(&metaevent);
{
   long              alarm_number;
   char              *alarm_name;
   unsigned long     alarm_source_objectID;
   char              *alarm_source_object_name;

   gc_AlarmNumber(metaeventp, &alarm_number);
                  // Will be of type TYPE_LAN_DISCONNECT = 0x01
                  // or TYPE_LAN_DISCONNECT + 0x10 (LAN connected).
   gc_AlarmName(metaeventp, &alarm_name);
                  // Will be  "Lan Cable Disconnected" or "Lan cable connected".
   gc_AlarmSourceObjectID(metaeventp, &alarm_source_objectID);
                  // Will usually be = 7.
   gc_AlarmSourceObjectName(metaeventp, &alarm_source_object_name)
                  // Will be "IPCCLIBAsoId"
   printf("Alarm %s (0x%lx) occurred on ASO %s (%d)", alarm_name, alarm_number,
           alarm_source_object_name, (int) alarm_source_objectID);
}
```

Building Global Call IP Applications 5

This chapter describes the IP-specific header files and libraries required when building applications.

Note: For more information about building applications, see the *Global Call API Programming Guide*.

5.1 Header Files

When compiling Global Call applications for the IP technology, it is necessary to include the following header files in addition to the standard Global Call header files, which are listed in the *Global Call API Library Reference* and *Global Call API Programming Guide*:

gcip.h
IP-specific data structures

gcip_defs.h
IP-specific type definitions, error codes and IP-specific parameter set IDs and parameter IDs

gccfgparm.h
Global Call type definitions, configurable parameters in the Global Call library and generic parameter set IDs and parameter IDs

gcipmlib.h
for Quality of Service (QoS) features

5.2 Required Libraries

When building Global Call applications for the IP technology, it is not necessary to link any libraries other than the standard Global Call library, *libgc.lib*. Other libraries, including IP-specific libraries, are loaded automatically.

5.3 Required System Software

The Intel® NetStructure™ Host Media Processing software must be installed on the development system. See the Software Installation Guide for your HMP release for further information.

Debugging Global Call IP Applications 6

This chapter provides information about debugging Global Call IP applications:

6.1 Debugging Overview

The Global Call IP Call Control Library uses the RTF (Runtime Tracking Facility) system that is used by other Intel telephony software libraries to write underlying call control library and stack information to a consolidated log file while an application is running. This information can help trace the sequence of events and identify the source of a problem. This information is also useful when reporting problems to technical support personnel.

All libraries and software modules that use RTF write their messages to a single, consolidated log file, with the default name *rftlog.txt*. The log file may optionally have a date and time stamp appended to the filename; for example, *rtflog01052005-13h24m19.923s*. When compared to the multiple independent log files used in previous implementations of the IP Call Control library, the consolidated log file has the advantage of clearly showing the time relationship of events associated with different software modules without requiring developers to correlate event time stamps.

Note: The SIP stack may also generate its own log file named *sdplog.txt* to capture any parsing errors that may occur.

The RTF facility allows developers to configure which events are written to the log file based on the importance of the event and the specific software module generating the event. All logging configuration for all libraries and modules that use RTF (not just the IP Call Control Library) is contained in a single, consolidated configuration file. This is in contrast to previous Global Call IP library implementations which used multiple configuration files for the library and the two IP protocol stacks.

The RTF facility uses the following entities to control which debug print statements are written to the log file:

module

An RTF module corresponds to a library or software module that has internal RTF APIs incorporated into its source code. Three separate RTF modules are used by the IP Call Control library:

- gc_h3r – call control, signal handler, and signal adaptation layer software modules
- sip_stack – SIP protocol stack
- h323_stack – H.323 protocol stack

client

An entity for identifying a device, component, or function that is to be traced by the RTF. The RTF modules for the IP Call Control library include a large number of client entities to provide a high degree of control over what statements are written to the log file; these clients are listed in the following sections which describe how to configure the logging facility.

label

An attribute associated with a trace statement to categorize the type or level of the information and to determine whether the statement is written to the log file. Labels are handled as independent entities and must be enabled or disabled individually; this is in contrast to the previous IP Call Control library logging implementation, where it was possible to enable log output for multiple statement levels collectively. Different RTF modules use different subsets of the overall RTF label set; the labels used for the IP Call Control library include only Error, Warning, and Debug.

6.2 Configuring the Logging Facility

The following topics provide information about how the user can customize the information written into the log file by the Global Call IP library:

- Configuration File Overview
- Configuring the gc_h3r Logging Module
- Configuring SIP Stack Logging
- Configuring H.323 Stack Logging

6.2.1 Configuration File Overview

This section describes how the common RTF configuration file is organized and what configuration is set up in the default configuration file that is supplied with the release software. The default configuration file may be named *RtfConfig.xml* or it may have an OS-specific name as appropriate to the specific release (i.e., *RtfConfigWin.xml* or *RtfConfigLinux.xml*); for simplicity, this document will only refer to the generic name. The entries in this configuration file conform to XML syntax rules.

Global Section

The global section of the *RtfConfig.xml* file contains one or more “GLabel” elements, which are used to globally enable logging of trace statements that are mapped to that RTF label. Globally enabling or disabling a label affects all RTF modules, but the global setting may be overridden locally.

The default *RtfConfig.xml* file globally enables the Error label, so that all error statements from all RTF modules will be logged unless disabled locally. The statement that globally enables the Error label is:

```
<GLabel name="Error" state="1"/>
```

Module Sections

The *RtfConfig.xml* file contains a number of module sections, each of which controls the logging of trace statements for a specific RTF module. Three RTF modules apply to the IP Call Control library: gc_h3r, h323_stack, and sip_stack.

Each module section begins with a <Module> tag (with name and state attributes) and ends with a </Module> tag. Between these two tags, the configuration file contains one or more “MLabel” elements to locally enable or disable logging of the RTF labels that are used by the specific module. The behavior of the “MLabel” elements for each of the RTF modules for the IP Call Control library are described in the following sections of this chapter.

Client Entries

In addition to “MLabel” elements, a module section may also contain a number of “MClient” elements for any clients that are defined within the module. Each of the three of the RTF modules for the IP Call Control library include a number of MClient elements, as described in the following sections of this chapter.

6.2.2 Configuring the gc_h3r Logging Module

The gc_h3r module controls logging of error and debug statements that related to the call control, signal handling, and signal adaptation layer software modules of the IP Call Control library. These statements were logged to the *gc_h3r.log* file in previous implementations.

The RTF gc_h3r module supports three user-maskable RTF labels: Error, Warning, and Debug. This is in contrast to the previous non-RTF implementation of the GC_H3R module, which used six debug levels. The old levels are mapped to the new labels as follows:

RTF Label (and default state)	Old GC_H3R Debug Levels
Error (globally enabled)	LEVEL_ERROR
Warning (locally enabled)	LEVEL_WARNING
Debug (locally disabled)	LEVEL_INFO, LEVEL_INFO_EXT, LEVEL_ALL

In addition to the five GC_H3R debug levels that are mapped to RTF labels, there is an additional level, LEVEL_SPECIAL, which is not mapped to an RTF label and is therefore non-maskable. Statements marked with LEVEL_SPECIAL are always printed to the log file.

The Error label is normally enabled globally. The Warning label is normally enabled locally, on the module level. The Debug label is enabled and disabled on the module level, and if the label is enabled the logging of these statements is controllable on an individual client basis.

The cg_h3r module in the *RtfConfig.xml* file begins with the statement:

```
<Module name="gc_h3r" state="1">
```

Following this statement are “MLabel” statements to set the local state of the Warning and Debug labels. In the default *RtfConfig.xml* file, the Warning label is enabled (state="1") and the Debug label is disabled (state="0").

```
<MLabel name="Warning" state="1"/>
<MLabel name="Debug" state="0"/>
```

In the gc_h3r module, the “MLabel” statement for the Warning label enables or disables the logging of all statements from the gc_h3r module that have LEVEL_WARNING in them regardless of the state settings of the “MClient” elements. The “MLabel” statement for the Debug label, on the other hand, interacts with the state settings of the “MClient” elements. Setting the state of the Debug label to "0" disables all statements containing LEVEL_INFO, LEVEL_INFO_EXT, or LEVEL_ALL, regardless of the MClient states. But setting the state of the Debug label to "1" only enables these statements for software modules that have their client state to "1". By enabling only the client modules are of interest in a given debug process, users can avoid the very large output that would result if all low-level statements from all gc_h3r software modules are logged.

Note: Enabling the Debug label while all of the gc_h3r clients are set to the enabled state may produce a very large log file and may cause significant loading of the CPU.

The “MClient” statements for each software module in the gc_h3r module follow the “MLabel” statements in the *RtfConfig.xml* file. The “MClient” statements are divided into four groups which correspond to four functional groups covered by this logging module. The prefixes of the client names also reflect this four-part grouping. A typical “MClient” statement looks like the following:

```
<MClient name="SH_CRN" state="1"/>
```

The following list gives the names and basic descriptions of the RTF clients in the GC_H3R module along with the corresponding module names that were used in the previous, non-RTF implementation of GC_H3R logging.

SH_CRN (formerly M_CRN)
Sharon Call Reference Number

SH_MGR (formerly M_SH_MAN)
Sharon Manager

SH_LD (formerly M_LD)
Sharon Line Device

SH_MEDIA (formerly M_MEDIA)
Sharon Media

SH_PDL (formerly M_PDL)
Sharon Platform Dependent Layer

SH_PACKER (formerly M_PACKER)
Sharon Packer

SH_DBASE (formerly M_SH_DB)
Sharon Database

SH_DECODER (formerly M_SH_DEC)
Sharon Decoder

SH_ENCODER (formerly M_SH_ENC)
Sharon Encoder

SH_IPC (formerly M_SH_IPC)
Sharon Inter-Process Communication

SH_UNPACK (formerly M_SH_UNPACK)
Sharon Unpacker

SH_BOARD (formerly M_BOARD)
Sharon Board Device.

SH_MONITOR (formerly M-MON)
Sharon Manager (host LAN monitor)

H323_SIG_MGR (formerly M_SIG_MAN)
H.323 Signal Adaptation Layer (Sigal) Manager

H323_CALL_MGR (formerly M_CALL_MAN)
H.323 Call Manager

H323_SIGNAL (formerly M_SIGNAL)
H.323 Signaling

H323_CONTROL (formerly M_CONTROL)
H.323 Control

H323_CH_MGR (formerly M_CHAN_MAN)
H.323 Channel Manager

H323_CHANNEL (formerly M_CHAN)
H.323 Channel

H323_IE (formerly M_IE)
H.323 Information Elements

H323_SIG_DEC (formerly M_SIG_DEC)
H.323 Signal Adaptation Layer Decoder

H323_SIG_ENC (formerly M_SIG_ENC)
H.323 Signal Adaptation Layer Encoder

H323_SIG_IPC (formerly M_SIG_IPC)
H.323 Inter-Process Communication

H323_RAS (formerly M_RAS)
H.323 Registration and Administration

H323_CAPS (formerly M_CAPS)
H.323 Capabilities

SIP_SIGAL (formerly M_S_SIGAL)
SIP Signal Adaptation Layer (Sigal)

SIP_SALL_MGR (formerly M_S_CALLM)
SIP Call Manager

SIP_SIGNAL (formerly M_S_SIGNL)
SIP Signaling

SIP_CH_MGR (formerly M_S_CHMGR)
SIP Channel Manager

SIP_IE (formerly M_SIP_IE)
SIP Information Elements

SIP_CAPS (formerly M_SIP_CAP)
SIP Capabilities

SIP_SIG_DEC (formerly M_SIP_DEC)
SIP Signal Adaptation Layer Decoder

SIP_SIG_ENC (formerly M_SIP_ENC)
SIP Signal Adaptation Layer Encoder

SIP_IPC (formerly M_SIP_IPC)
Inter-Process Communication

SIP_INFO (formerly M_INFO)
SIP Information

SIP_REFER (formerly M_REFER)
SIP Refer

SIP_PRACK (formerly M_PRACK)
SIP Protocol Acknowledgement

SIP_AUTHENT (formerly M_AUTHENT)
SIP Authenticator

SIP_SUBSYS (formerly M_S_SUBSM)
SIP Subsystem

COM_MEMMGR (formerly M_MEMMGR)
Common Memory Manager

COM_MIME (formerly M_MIME)
Common Mime

COM_R_MGR (formerly M_R_MGR)
Common “R” Manager

COM_MR_MGR (formerly M_MR_MGR)
Common “MR” Manager

6.2.3 Configuring SIP Stack Logging

The sip_stack RTF module controls logging of debug statements that relate to the SIP protocol stack used by the IP Call control library. In previous implementations, this logging was configured via the *gc_h3r.cfg* file and the statements were logged to the file *gc_h3r.log*.

Note: The SIP stack may also generate its own log file named *sdplog.txt* to capture any parsing errors that occur.

The sip_stack module supports two user-maskable RTF labels: Error and Debug. This is in contrast to the previous non-RTF implementation of the GC_H3R module, which used five bit-encoded debug levels. The old levels are mapped to the new labels as follows:

RTF Label (and default state)	Old SIP Debug Levels in GC_H3R
Error (globally enabled)	EXCEP, ERROR, WARN
Debug (locally disabled)	INFO, DEBUG

The Error label is normally enabled globally. The Debug label is enabled and disabled on the module level, and if the label is enabled the logging of these statements is controllable on an individual client basis. The state of the Warning label has no effect on the sip_stack module.

The sip_stack module in the *RtfConfig.xml* file begins with the statement:

```
<Module name="sip_stack" state="1">
```

Following this statement is an “MLabel” statement to set the local state of the Debug label, which is disabled (state="0") in the default *RtfConfig.xml* file:

```
<MLabel name="Debug" state="0"/>
```

The “MLabel” statement for the Debug label interacts with the state settings of the “MClient” elements to enable or disable logging from the individual software modules of the SIP protocol stack. Setting the state of the Debug label to "0" disables all debug statements from the SIP stack, regardless of the states of the individual RTF clients. Setting the state of the Debug label to "1" enables logging of debug statements for any stack modules that have their RTF client state to "1".

Note: Enabling the Debug label while all of the sip_stack clients are set to the enabled state may produce a very large log file and may cause significant loading of the CPU.

The “MClient” statements for each software module in the sip_stack module follow the “MLabel” statement in the *RtfConfig.xml* file. A typical “MClient” statement in the *RtfConfig.xml* file looks like the following, which enables logging for the MESSAGE client if the Debug label is enabled:

```
<MClient name="MESSAGE" state="1"/>
```

The names of the RTF clients in the sip_stack module (along with the module names used in the previous GC_H3R logging implementation) include the following:

- MESSAGE (formerly RvSipStack_Message)
- TRANSPORT (formerly RvSipStack_Transport)
- TRANSACTION (formerly RvSipStack_Transaction)
- CALL (formerly RvSipStack_Call)
- PARSER (formerly RvSipStack_Parser)
- STACK (formerly RvSipStack_Stack)
- MSG BUILDER (formerly RvSipStack_MsgBuilder)
- AUTHENTICATOR (formerly RvSipStack_Authenticator)
- REG CLIENT (formerly RvSipStack_RegClient)
- SUBSCRIPTION

6.2.4 Configuring H.323 Stack Logging

The “h323_stack” RTF module controls logging of debug statements that relate to the H.323 protocol stack used by the IP Call control library. In previous implementations, this logging was configured via the *rvtele.ini* file and the statements were logged to the file *rvtsp1.log*.

The h323_stack RTF module uses a single label, namely Debug. The states of the Error and Warning labels have no effect on the h323_stack module.

The h323_stack module in the *RtfConfig.xml* file begins with the statement:

```
<Module name="h323_stack" state="1">
```

Following this statement is an "MLabel" statement to set the local state of the Debug label, which is disabled (state="0") in the default *RtfConfig.xml* file:

```
<MLabel name="Debug" state="0"/>
```

The "MLabel" statement for the Debug label interacts with the state settings of the "MClient" elements to enable or disable logging from the individual software modules of the H.323 protocol stack. Setting the state of the Debug label to "0" disables all debug statements from the H.323 stack, regardless of the states of the individual RTF clients. Setting the state of the Debug label to "1" enables logging of debug statements for any stack modules that have their RTF client state to "1".

Note: Enabling the Debug label while all of the h323_stack clients are set to the enabled state may produce a huge log file and may cause heavy loading of the CPU.

The "MClient" statements for each software module in the h323_stack module follow the "MLabel" statement in the *RtfConfig.xml* file. A typical "MClient" statement in the *RtfConfig.xml* file looks like the following, which enables logging for the EMA stack module if the Debug label is also enabled:

```
<MClient name="EMA" state="1"/>
```

The names of the RTF clients in the h323_stack module include the following (the † symbol marks the clients that are most commonly used in debugging):

- EMA
- MEMORY
- RA
- CAT
- CM †
- CMAPI †
- CMAPICB †
- CMERR †
- TPKTCHAN †
- CONFIG †
- APPL
- FASTSTART †
- VT
- UNREG
- RAS †
- UDPCHAN
- TCP
- TRANSPORT
- ETIMER

- PER †
- PERERR †
- Q931†
- Q931ERR
- LI
- TIMER
- ANNEXE
- SSEERR
- SSEAPI
- SSEAPICB
- SUPS
- SSCHAN

IP-Specific Function Information 7

Certain Global Call functions have additional functionality or perform differently when used with IP technology. The generic function descriptions in the *Global Call API Library Reference* do not contain detailed information for any specific technology. Detailed information in terms of the additional functionality or the difference in performance of those functions when used with IP technology is contained in this chapter. The information provided in this guide therefore must be used in conjunction with the information presented in the *Global Call API Library Reference* to obtain the complete information when developing Global Call applications that use IP technology. IP-specific variances are described in the following topics:

7.1 Global Call Functions Supported by IP

The following is a full list of the Global Call functions that indicates the level of support when used with IP technology. The list indicates whether the function is supported, not supported, or supported with variances.

gc_AcceptCall( )
Supported in asynchronous mode only with variances described in Section 7.3.1, "gc_AcceptCall( ) Variances for IP", on page 296

gc_AcceptInitXfer( )
Supported with variances described in Section 7.3.2, "gc_AcceptInitXfer( ) Variances for IP", on page 296

gc_AcceptXfer( )
Supported with variances described in Section 7.3.3, "gc_AcceptXfer( ) Variances for IP", on page 297

gc_AlarmName( )
Supported

gc_AlarmNumber( )
Supported

gc_AlarmNumberToName( )
Supported

gc_AlarmSourceObjectID( )
Supported

gc_AlarmSourceObjectIDToName( )
Supported

gc_AlarmSourceObjectName( )
Supported

gc_AlarmSourceObjectNameToID( )
Supported

gc_AnswerCall( )
Supported in asynchronous mode only with variances described in Section 7.3.4, "gc_AnswerCall( ) Variances for IP", on page 298

gc_AttachResource( )
Supported in asynchronous mode only

gc_BlindTransfer( )
Not supported

gc_CallAck( )
Supported in asynchronous mode only with variances described in Section 7.3.5, "gc_CallAck( ) Variances for IP", on page 299

gc_CCLibIDToName( )
Supported

gc_CCLibNameToID( )
Supported

gc_CCLibStatus( )
Supported, but deprecated. Use **gc_CCLibStatusEx( )**.

gc_CCLibStatusAll( )
Supported, but deprecated. Use **gc_CCLibStatusEx( )**.

gc_CCLibStatusEx( )
Supported

gc_Close( )
Supported

gc_CRN2LineDev( )
Supported

gc_Detach( )
Supported in asynchronous mode only

gc_DropCall( )
Supported in asynchronous mode only with variances described in Section 7.3.6, "gc_DropCall( ) Variances for IP", on page 300

gc_ErrorInfo( )
Supported

gc_ErrorValue( )
Supported, but deprecated. Use **gc_ErrorInfo( )**.

gc_Extension( )
Supported in asynchronous mode only with variances described in Section 7.3.7, “gc_Extension( ) Variances for IP”, on page 300

gc_GetAlarmConfiguration( )
Supported

gc_GetAlarmFlow( )
Supported

gc_GetAlarmParm( )
Supported with variances described in Section 7.3.8, “gc_GetAlarmParm( ) Variances for IP”, on page 302

gc_GetAlarmSourceObjectList( )
Supported

gc_GetAlarmSourceObjectNetworkID( )
Supported

gc_GetCallInfo( )
Supported with variances described in Section 7.3.9, “gc_GetCallInfo( ) Variances for IP”, on page 303

gc_GetCallState( )
Supported

gc_GetCRN( )
Supported

gc_GetCTInfo( )
Supported with variances described in Section 7.3.10, “gc_GetCTInfo( ) Variances for IP”, on page 306

gc_GetLineDev( )
Supported

gc_GetMetaEvent( )
Supported

gc_GetMetaEventEx( )
Supported (Windows extended asynchronous programming model only)

gc_GetResourceH( )
Supported with variances described in Section 7.3.11, “gc_GetResourceH( ) Variances for IP”, on page 306

gc_GetUsrAttr( )
Supported

gc_GetVer( )
Supported

gc_GetXmitSlot( )
Supported with variances described in Section 7.3.12, “gc_GetXmitSlot( ) Variances for IP”, on page 306

gc_InitXfer( )
Supported with variances described in Section 7.3.13, “gc_InitXfer( ) Variances for IP”, on page 306

gc_InvokeXfer( )
Supported with variances described in Section 7.3.14, “gc_InvokeXfer( ) Variances for IP”, on page 307

gc_LinedevToCCLIBID( )
Supported

gc_Listen( )
Supported with variances described in Section 7.3.15, “gc_Listen( ) Variances for IP”, on page 311

gc_MakeCall( )
Supported in asynchronous mode only with variances described in Section 7.3.16, “gc_MakeCall( ) Variances for IP”, on page 311

gc_OpenEx( )
Supported with variances described in Section 7.3.17, “gc_OpenEx( ) Variances for IP”, on page 326

gc_RejectInitXfer( )
Supported with variances described in Section 7.3.18, “gc_RejectInitXfer( ) Variances for IP”, on page 327

gc_RejectXfer( )
Supported with variances described in Section 7.3.19, “gc_RejectXfer( ) Variances for IP”, on page 328

gc_ReleaseCallEx( )
Supported with variances described in Section 7.3.20, “gc_ReleaseCallEx( ) Variances for IP”, on page 329

gc_ReqService( )
Supported in asynchronous mode only with variances described in Section 7.3.21, “gc_ReqService( ) Variances for IP”, on page 329

gc_ResetLineDev( )
Supported in asynchronous mode only

gc_RespService( )
Supported in asynchronous mode only with variances described in Section 7.3.22, “gc_RespService( ) Variances for IP”, on page 333

gc_ResultInfo( )
Supported

gc_SetAlarmConfiguration( )
Supported

gc_SetAlarmFlow( )
Supported

gc_SetAlarmNotifyAll( )
Supported

gc_SetAlarmParm( )
Supported with variances described in Section 7.3.23, “gc_SetAlarmParm( ) Variances for IP”, on page 333

gc_SetAuthenticationInfo( )
New IP-specific function; see page 279 for complete information

gc_SetConfigData( )
Supported in asynchronous mode only with variances described in Section 7.3.24, “gc_SetConfigData( ) Variances for IP”, on page 334

gc_SetUserInfo( )
Supported with variances described in Section 7.3.25, “gc_SetUserInfo( ) Variances for IP”, on page 337

gc_SetUsrAttr( )
Supported

gc_Start( )
Supported with variances described in Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340

gc_Stop( )
Supported

gc_UnListen( )
Supported with variances described in Section 7.3.27, “gc_UnListen( ) Variances for IP”, on page 343

gc_util_copy_parm_blk( )
New supported function; see page 282 for full details

gc_util_delete_parm_blk( )
Supported

gc_util_find_parm( )
Supported

gc_util_find_parm_ex( )
New supported function; see page 284 for full details

gc_util_insert_parm_ref( )
Supported

gc_util_insert_parm_ref( )
New supported function; see page 287 for full details

gc_util_insert_parm_val( )
Supported

gc_util_next_parm( )
Supported

gc_util_next_parm_ex( )
New supported function; see page 290 for full details

gc_WaitCall( )
Supported in asynchronous mode only

7.2 IP-Specific Global Call Functions

The following API reference pages describe Global Call functions that are specific to the use of IP technology:

- **gc_SetAuthenticationInfo( )**
- **gc_util_copy_parm_blk( )**
- **gc_util_find_parm_ex( )**
- **gc_util_insert_parm_ref_ex( )**
- **gc_util_next_parm_ex( )**
- **IP_VIRTBOARD**
- **INIT_IPCCLIB_START_DATA( )**

Note: The new **gc_util_...** functions are backwards compatible with existing **gc_util_...** functions and may be used with any Global Call technology, but IP technology (specifically the sending and retrieving of SIP message headers) is currently the only situation where these functions *must* be used to support parameter data longer than 255 bytes.

gc_SetAuthenticationInfo( )

Name:	int gc_SetAuthenticationInfo(target_type, target_id, infoparmblkp)	
Inputs:	int target_type	• type of target object (virtual board)
	long target_id	• target object ID
	GC_PARM_BLKP infoparmblkp	• pointer to CG_PARM_BLK with user information
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Mode:	synchronous	

■ Description

The **gc_SetAuthenticationInfo( )** function is used to configure or remove authentication information on an IPT virtual board. This is the only Global Call function that can be used to set this information; the generic Global Call functions **gc_SetConfigData( )** and **gc_SetUserInfo( )** functions cannot be used for this IP-specific configuration operation.

This function should be called before using any Global Call function that sends a SIP request that may provoke a 401/407 response. A 401/407 response to any SIP request that was sent before authentication is configured causes the request to be terminated (with the reason code IPEC_SIPReasonStatus401Unauthorized or IPEC_SIPReasonStatus407ProxyAuthenticationRequired), and Global Call will not attempt to re-send the request.

Parameter	Description
target_type	specifies the type of target object; must be set to GCTGT_CCLIB_NETIF.
target_id	specifies the virtual board ID that the authentication information applies to
infoparmblklp	points to a GC_PARM_BLK structure that contains the authentication information. The parm block contains one or more parameters that use the IPSET_CONFIG set ID and IPPARM_AUTHENTICATION_CONFIGURE or IPPARM_AUTHENTICATION_REMOVE as the parameter ID. In either case, the parameter data is an IP_AUTHENTICATION data structure that contains an authentication quadruplet of {realm, identity, username, password}.

An IPSET_CONFIG/IPPARM_AUTHENTICATION_CONFIGURE parameter in **infoparmblkp** adds a new authentication quadruplet to the Global Call database if the realm and identity strings in the IP_AUTHENTICATION structure are unique. If both the realm and identity strings match a quadruplet that already exists, the existing username and password are overwritten with the new strings. If the identity field in the IP_AUTHENTICATION structure is an empty string, the function will set the specified username and password as the defaults for the specified realm.

An IPSET_CONFIG/IPPARM_AUTHENTICATION_REMOVE parameter in **infoparmblkp** removes the existing authentication quadruplet that matches the realm and identity strings that are specified in the IP_AUTHENTICATION structure. When removing a quadruplet, the username and password elements in the IP_AUTHENTICATION structure are ignored.

Cautions

- The **gc_SetAuthenticationInfo( )** function can only be called on a virtual board device.
- If the CG_PARM_BLK contains multiple parameter elements with the same realm/identity pair in their IP_AUTHENTICATION structures, all of those parameters are ignored except for the one that is last in the GC_PARM_BLK.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

Possible errors include:

IPERR_BAD_PARM
: returned if any of the string pointers in an IP_AUTHENTICATION structure is NULL or if there is any other invalid parameter

IPERR_UNAVAILABLE
: returned when the realm/identity does not exist in the Global Call database when the application attempts to remove the quadruplet

IPERR_UNSUPPORTED
: returned when the function is called on a line device or CRN rather than a virtual board

Examples

The following code example illustrates how to add or modify a digest authentication quadruplet.

```
#include <gcip.h>
#include <gclib.h>

/* This example adds or modifies the quadruplet with realm "example.com" and
 * identity "sip:bob@example.com". If this realm/identity do not exist on this
 * virtual board, this quadruplet will be added. If this realm/identity exist
 * already, it will be override by this quadruplet.
 */

void configureAuthQuadruplet (long boardDev)
{
   GC_PARM_BLK *parmblkp = NULL;
   char realm[] = "example.com";
   char identity[] = "sip:bob@example.com";
   char username[] = "bob";
   char password [] = "password1";
   IP_AUTHENTICATION authentication;
   INIT_IP_AUTHENTICATION (&authentication);
   authentication.realm = realm;
   authentication.identity = identity;
   authentication.username = username;
   authentication.password = password;

   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_CONFIG,
                           IPPARM_AUTHENTICATION_CONFIGURE,
                           (unsigned char)(sizeof(IP_AUTHENTICATION)),
                           &authentication);

   gc_SetAuthenticationInfo(GCTGT_CCLIB_NETIF, boardDev, parmblkp);

   gc_util_delete_parm_blk(parmblkp);
}
```

The following code example illustrates how to remove a digest authentication quadruplet.

```
#include <gcip.h>
#include <gclib.h>

/* This example deletes the quadruplet with realm "example.com" and
 * identity "sip:bob@example.com".
 */

void removeAuthQuadruplet (long boardDev)
{
   GC_PARM_BLK *parmblkp = NULL;
   char realm[] = "example.com";
   char identity[] = "sip:bob@example.com";

   IP_AUTHENTICATION authentication;
   INIT_IP_AUTHENTICATION (&authentication);

   authentication.realm = realm;
   authentication.identity = identity;

   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_CONFIG,
                           IPPARM_AUTHENTICATION_REMOVE,
                           (unsigned char)(sizeof(IP_AUTHENTICATION)),
                           &authentication);

   gc_SetAuthenticationInfo(GCTGT_CCLIB_NETIF, boardDev, parmblkp);

   gc_util_delete_parm_blk(parmblkp);
}
```

■ See Also

None.

gc_util_copy_parm_blk( )

Name:	int gc_util_copy_parm_blk(parm_blkpp, parm_blkp)	
Inputs:	GC_PARM_BLKP* parm_blkpp	• pointer to the address of the new GC_PARM_BLK
	GC_PARM_BLKP parm_blkp	• pointer to a valid GC_PARM_BLK to be copied
Returns:	GC_SUCCESS if successful GC_FAIL if unsuccessful	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_copy_parm_blk( )** function copies the specified GC_PARM_BLK.

This function **must** be used to copy any GC_PARM_BLK that contains any setID/parmID pairs that can have data that is potentially larger than 255 bytes. This function can be used for any GC_PARM_BLK, regardless of whether it contains setID/parmID pairs that support parameter data lengths greater than 255 bytes.

Note: The only Global Call parameter that currently supports data longer than 255 bytes is IPSET_SIP_MSGINFO / IPPARM_SIP_HDR, which is used for SIP message headers.

Parameter	Description
parm_blkpp	pointer to the address of the new GC_PARM_BLK that the specified parm block will be copied to; **must** be set to NULL
parm_blkp	points to a valid, existing GC_PARM_BLK to be copied

■ Cautions

To avoid a memory leak, any GC_PARM_BLK created must eventually be deleted using the **gc_util_delete_parm_blk( )** function.

■ Errors

If this function returns GC_ERROR(-1) to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

■ Example

```
#include "gclib.h"
#include "gcip.h"
```

```
void process_event(void)
{
   METAEVENT  metaevent;
   GC_PARM_BLKP my_blkp = NULL;

   if(gc_GetMetaEvent(&metaevent) != GC_SUCCESS)
   {
      /* process error */
   }

   Switch(metaevent.evttype)
   {
      case GCEV_OFFERED:
         /* make a copy of the parm blk */
         if(metaevent.extevtdatap)
         {
            if ( gc_util_copy_parm_blk( &my_blkp,(GC_PARM_BLKP)(metaevent.extevtdatap))
                 != GC_SUCCESS )
            {
               /* Process error */
            }
         }
      ......
   }
   ......
}
```

■ See Also

- **gc_util_delete_parm_blk( )** (in *Global Call API Library Reference*)

gc_util_find_parm_ex( )

Name:	int gc_util_find_parm_ex(parm_blk, setID, parmID, parm)	
Inputs:	GC_PARM_BLKP parm_blk	• pointer to GC_PARM_BLK to search for the parameter
	unsigned long setID	• parameter set ID of parameter to be found
	unsigned long parmID	• parameter ID of parameter to be found
	GC_PARM_DATA_EXTP parm	• pointer to a valid GC_PARM_DATA_EXT structure that identifies where in the parm block to start searching
Outputs:	GC_PARM_DATA_EXTP parm	• if successful, pointer to a GC_PARM_DATA_EXT structure that contains the ID and value data for the specified parameter
Returns:	GC_SUCCESS if successful EGC_NO_MORE_PARMS if no more parameters exist in GC_PARM_BLK GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_find_parm_ex**( ) function is used to find a parameter of a particular type in a GC_PARM_BLK and retrieve the parameter data into a GC_PARM_DATA_EX structure.

This function **must** be used instead of the similar **gc_util_find_parm**( ) function if the parameter data can potentially exceed 255 bytes. This function is backward compatible and can be used instead of **gc_util_find_parm**( ) for any GC_PARM_BLK, regardless of whether the parameter block contains setID/parmID pairs that support data lengths greater than 255 bytes.

Note: The only Global Call parameter that currently supports data longer than 255 bytes is IPSET_SIP_MSGINFO / IPPARM_SIP_HDR, which is used for SIP message headers.

The **gc_util_find_parm_ex**( ) function can be used to determine whether a particular parameter exists, or to retrieve a particular parameter, or both. If the specified parameter is found in the GC_PARM_BLK, the function fills in the GC_PARM_DATA_EX structure with the parameter data and returns GC_SUCCESS. If the parameter does not exist in the GC_PARM_BLK, or if no more parameters of the specified type are found, the function returns EGC_NO_MORE_PARMS.

To search from the beginning of the GC_PARM_BLK, initialize the GC_PARM_DATA_EXT structure (**parm**) by using **INIT_GC_PARM_DATA_EXT(parm)** before calling **gc_util_find_parm_ex**( ). If the structure pointed to by **parm** contains parameter information that was retrieved in a previous call to this function, the function will begin its search at that parameter rather than the beginning of the parameter block.

Parameter	Description
parm_blk	points to a valid GC_PARM_BLK that will be searched for a parameter of the specified type
setID	set ID of the parameter to be found
parmID	parameter ID of the parameter to be found
parm	points to a valid GC_PARM_DATA_EXT provided by the application. If a pointer to a newly initialized structure is passed in the function call, the function searches from the beginning of the GC_PARM_BLK; if the structure contains data from a previously found parameter, the function searches from that parameter onward. When the function completes successfully, the structure is updated to contain retrieved information for the parameter that was found.

Cautions

Unlike the similar **gc_util_find_parm( )** function, the **parm** pointer used in this function *cannot* be used to update the parameter itself because it points to a data structure that is in the application's memory rather than a location in the GC_PARM_BLK itself.

Errors

If this function returns GC_ERROR to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the "Error Handling" section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

Example

```
#include "gclib.h"
#include "gcip.h"

void search_parm_block(GC_PARM_BLKP parm_blkp)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To search from the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);

   /* loop to retrieve all of the parameters and associated data in the
    * GC_PARM_BLK that match the set_ID/parm_ID pair for SIP header fields.
    */
   while ( GC_SUCCESS == (ret = gc_util_find_parm_ex(parm_blkp, IPSET_SIP_MSGINFO,
                                                     IPPARM_SIP_HDR, &parm_data_ext)) )
   {
      /* process GC_PARM_DATA_EXT structure */
      .
      .
      .
   }
```

```
    /* Check for error */
    if ( GC_ERROR == ret)
    {
        /* process error */
    }

    .
    .
    .
}
```

See Also

- **gc_util_next_parm_ex( )**

gc_util_insert_parm_ref_ex( )

Name:	int gc_util_insert_parm_ref_ex(parm_blkpp, setID, parmID, data_size, datap)	
Inputs:	GC_PARM_BLKP *parm_blkpp	• pointer to the address of a valid GC_PARM_BLK
	unsigned long setID	• set ID of parameter to be inserted
	unsigned long parmID	• parm ID of parameter to be inserted
	unsigned long data_size	• size in bytes of the parameter data
	void *datap	• pointer to the parameter data
Returns:	GC_SUCCESS if successful GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

Description

The **gc_util_insert_parm_ref_ex**( ) function inserts a parameter into a GC_PARM_BLK data structure by reference.

The **gc_util_insert_parm_ref_ex**( ) function **must** be used rather than the similar **gc_util_insert_parm_ref( )** function whenever the parameter data exceeds 255 bytes in length. The **gc_util_insert_parm_ref_ex**( ) function is backwards compatible and can be used with any setID/parmID pair regardless of whether that pair supports data lengths greater than 255 bytes.

Note: The only Global Call parameter that currently supports data longer than 255 bytes is IPSET_SIP_MSGINFO / IPPARM_SIP_HDR, which is used for SIP message headers.

A new GC_PARM_BLK can be created by inserting the first parameter with ***parm_blkpp** set to NULL. A parameter can be inserted in an existing GC_PARM_BLK by setting ***parm_blkpp** to the address of that block.

Note: Parameters are contained in the GC_PARM_BLK in the order in which they are inserted, and they will also be retrieved via the **gc_util_next_parm_ex( )** function in the same order.

Parameter	Description
parm_blkpp	points to the address of a valid GC_PARM_BLK where the parameter is to be inserted. Set ***parm_blkpp** to NULL to insert the parameter into a new block.
setID	parameter set ID of the parameter to be inserted
parmID	parameter ID of the parameter to be inserted

Parameter	Description
data_size	size, in bytes, of the data associated with this parameter
datap	points to the data associated with this parameter

■ Cautions

- To avoid a memory leak, any GC_PARM_BLK created must be deleted using the **gc_util_delete_parm_blk( )** function.
- Insertion of data that exceeds 255 bytes in length is only supported for specific setID/parmID pairs. Refer to the appropriate Global Call Technology Guide for information on maximum data length for each setID/parmID pair.

■ Errors

- If this function returns GC_ERROR to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the "Error Handling" section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.
- Attempting to insert data greater than 255 bytes in length using a setID/parmID pair that does not support large data sizes produces an error indication. In this situation, the **gc_ErrorInfo( )** function returns the value EGC_INVPARM.

■ Example

```
#include "gclib.h"
#include "gcip.h"

void SetHeader(void)
{
   GC_PARM_BLKP my_blkp = NULL;
   char* pChar = "Remote-Party_ID: This string can be greater than 255 bytes";

   /* Add 1 to strlen for null termination */
   unsigned long data_size = strlen(pChar) + 1;

   /* insert parm and associated data into the GC_PARM_BLK */
   if ( gc_util_insert_parm_ref_ex( &my_blkp, IPSET_SIP_MSGINFO, IPPARM_SIP_HDR, data_size,
                                    (void*)( pChar )) != GC_SUCCESS )
   {
      /* Process error */
   }

   /* At this point the application can overwrite the data pointed to by pChar. */
   pChar = NULL;

   /* Pass the parm block to GC */
   if ( gc_SetUserInfo( GCTGT_GCLIB_CRN, crn, &my_blkp, GC_SINGLECALL) != GC_SUCCESS )
   {
      /* Process error */
   }

   /* GC_PARM_BLK is no longer needed; delete the block */
   gc_util_delete_parm_blk( my_blkp );
}
```

■ See Also

- **gc_util_delete_parm_blk( )** (in *Global Call API Library Reference*)
- **gc_util_insert_parm_ref( )** (in *Global Call API Library Reference*)
- **gc_util_insert_parm_val( )** (in *Global Call API Library Reference*)

gc_util_next_parm_ex( )

Name:	int gc_util_next_parm_ex(parm_blk, parm)	
Inputs:	GC_PARM_BLKP parm_blk	• pointer to GC_PARM_BLK
	GC_PARM_DATA_EXTP parm	• pointer to structure identifying current parameter
Outputs:	GC_PARM_DATA_EXTP parm	• pointer to structure identifying next parameter
Returns:	GC_SUCCESS if successful EGC_NO_MORE_PARMS if no more parameters exist in the GC_PARM_BLK GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_next_parm_ext( )** function is used to retrieve the next parameter (relative to a specified current parameter) from a GC_PARM_BLK in the form of a GC_PARM_DATA_EX data structure. Calling this function repetitively and passing a pointer to the GC_PARM_DATA_EX structure that was returned by the previous call allows an application to sequentially retrieve all of the parameters in a GC_PARM_BLK.

This function **must** be used instead of **gc_util_next_parm( )** if the parameter data can potentially exceed 255 bytes. This function is backward compatible and can be used instead of **gc_util_next_parm( )** for any GC_PARM_BLK, regardless of whether the parameter block contains setID/parmID pairs that support data lengths greater than 255 bytes.

Note: The only Global Call parameter that currently supports data longer than 255 bytes is IPSET_SIP_MSGINFO / IPPARM_SIP_HDR, which is used for SIP message headers.

The **gc_util_next_parm_ex( )** function updates the data structure pointed to by the **parm** pointer and returns GC_SUCCESS if there is another parameter in the GC_PARM_BLK following the current parameter that was identified in the function call. If the current parameter identified by **parm** is the last parameter in the GC_PARM_BLK, the function returns EGC_NO_MORE_PARMS.

Parameter	Description
parm_blk	points to the valid GC_PARM_BLK structure where data is stored
parm	pointer to a valid GC_PARM_DATA_EXT structure provided by the application. If the pointer that is passed in the function call points to a structure that was just initialized with INIT_GC_PARM_DATA_EXT(parm), the function retrieves the first parameter in the GC_PARM_BLK. If the passed pointer points to a structure that contains data from a previously found parameter, the function retrieves the next parameter in the block (if any). When the function completes successfully, the GC_PARM_DATA_EXT structure is updated to contain retrieved information for the parameter.

■ Cautions

Unlike the similar **gc_util_next_parm( )** function, the **parm** pointer used in this function *cannot* be used to update the parameter itself because it points to a data structure that is in the application's memory rather than a location in the GC_PARM_BLK itself.

■ Errors

If this function returns GC_ERROR to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the "Error Handling" section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

■ Example

```
#include "gclib.h"
#include "gcip.h"

void process_parm_block(GC_PARM_BLKP pparm_blk)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To retrieve the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);

   /* Loop to retrieve all of the parameters and associated data from
    * the GC_PARM_BLK
    */insert parm by reference */
   while ( GC_SUCCESS == (ret = gc_util_next_parm_ex( pparm_blk, &parm_dat_ext)) )
   {
      /* Process set_ID/parm_ID pairs */
      switch(parm_data_ext.set_ID);
      {
         .
         .
         .
      }
   }
   /* Check for error */
   if ( GC_ERROR == ret )
   {
      /* Process error */
   }

   .
   .
   .
}
```

See Also

- **gc_util_find_parm_ex( )**

INIT_IP_VIRTBOARD( )

Name:	void INIT_IP_VIRTBOARD(pIpVb)	
Inputs:	IP_VIRTBOARD *pIpVb	• pointer to the structure to be initialized
Returns:	None	
Includes:	gcip.h	
Mode:	synchronous	

Description

The **INIT_IP_VIRTBOARD( )** function is used to initialize an IP_VIRTBOARD data structure, which contains configuration data for a specific virtual IPT board. This function must be called to initialize an IP_VIRTBOARD structure for each virtual board that will be defined by calling **INIT_IPCCLIB_START_DATA( )** before calling **gc_Start( )**.

After the structure is initialized, an application can overwrite any of the defeat values as appropriate to the specific requirements. Among the items controlled by the IP_VIRTBOARD structure and initialized by this function are:

- Maximum number of calls (total, H.323, and SIP)
- Local IP address and signaling ports for H.323 and SIP
- H.323 Terminal Type (default is Gateway)
- Enable access to H.323 message information fields (default is disabled)
- Enable call transfer supplementary service (default is disabled)
- Enable access to SIP message header fields and MIME-encoded message bodies (default is access disabled for both headers and MIME bodies)
- Enable and configure a SIP outbound proxy (default is disabled)
- Enable and configure TCP transport for SIP requests (default is disabled)
- Configure SIP request retry behavior (default enables all allowable retries)
- Enable application access to SIP OPTIONS requests (default is disabled)

Parameter	Description
pIpVb	points to the IP_VIRTBOARD data structure to be initialized. See IP_VIRTBOARD, on page 394, for information on the default values and optional values that may be after initialization.

Cautions

None.

Example

```
IP_VIRTBOARD ip_virtboard[2];
IPCCLIB_START_DATA ipcclibstart;
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
```

See Also

- **INIT_IPCCLIB_START_DATA( )**
- Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340

INIT_IPCCLIB_START_DATA( )

Name:	void INIT_IPCCLIB_START_DATA(pIpStData, numBoards, pIpVb)	
Inputs:	IPCCLIB_START_DATA *pIpStData	• pointer to the structure to be initialized
	unsigned char numBoards	• number of boards
	IP_VIRTBOARD *pIpVb	• pointer to an array of IP_VIRTBOARD structures
Returns:	None	
Includes:	gcip.h	
Mode:	synchronous	

Description

The **INIT_IPCCLIB_START_DATA( )** function is used to initialize the IPCCLIB_START_DATA data structure, which contains configuration information on the virtual IPT boards to be started via **gc_Start( )**.

Applications **must** use this function to initialize the IPCCLIB_START_DATA structure before calling **gc_Start( )**.

Parameter	Description
pIpStData	points to the IPCCLIB_START_DATA structure to be initialized
numBoards	the number of virtual IPT boards being defined (up to a maximum of 8)
pIpVb	points to an array of IP_VIRTBOARD data structures, one for each virtual IPT board being defined

Cautions

None.

Example

```
IP_VIRTBOARD ip_virtboard[2];
IPCCLIB_START_DATA ipcclibstart;
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
```

See Also

- **INIT_IP_VIRTBOARD( )**
- Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340

7.3 Global Call Function Variances for IP

Note: Except for **gc_Listen( )**, **gc_OpenEx( )**, **gc_ReleaseCallEx( )**, **gc_UnListen( )**, all Global Call functions that nominally support synchronous and asynchronous mode are supported only in asynchronous mode when using the IP technology.

The Global Call function variances that apply when using IP technology are described in the following sections. See the *Global Call API Library Reference* for generic (technology-independent) descriptions of the Global Call API functions.

7.3.1 gc_AcceptCall( ) Variances for IP

This function is only supported in asynchronous mode.

The **rings** parameter is ignored.

Variance for H.323

The **gc_AcceptCall( )** function is used to send the Q.931 ALERTING message to the originating endpoint.

Variance for SIP

The **gc_AcceptCall( )** function is used to send the 180 Ringing message to the originating endpoint.

7.3.2 gc_AcceptInitXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

Variance for H.323 (H.450.2)

Either the rerouting_num (of type char*) or rerouting_addrblkp (of type GCLIB_ADDRESS_BLK*) fields of the GC_REROUTING_INFO structure can be used to specify the rerouting address string to be signaled back to party A and its final destination to party B. The sub_address fields of the GCLIB_ADDRESS_BLK are ignored and not used.

Note: If both fields are used, the rerouting address string will be a concatenation of the information from both fields.

The GCEV_ACCEPT_INIT_XFER event is received by the application on the secondary/consultation call CRN once the transferred call is received as notified via the GCEV_OFFERED event.

If the call transfer is abandoned by parties A or B before the transfer is completed, the GCEV_ACCEPT_INIT_XFER_FAIL event is received with a CCLIB cause value of IPEC_H4502CTAbandon and a Global Call cause value of GCRV_CALLABANDONED.

If the CTT2 timer (20 seconds) expires before the transfer is completed, the GCEV_ACCEPT_INIT_XFER_FAIL event is received with a CCLIB cause value of IPEC_H450CTT2Timeout and a Global Call cause value of GCRV_TIMEOUT.

Variance for SIP

This function does not apply to SIP call transfer. In SIP, party A does not notify party C in advance of requesting an attended (supervised) transfer operation with **gc_InvokeXfer( )**, so there is no opportunity for party C to accept or reject the transfer at the initiation stage.

7.3.3 gc_AcceptXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

The **parmblkp** parameter is ignored for IP technology and should be set to NULL.

The **gc_AcceptXfer( )** function can be used at party B only after receiving a GCEV_REQ_XFER event. The application can obtain information on the rerouting number or address in a GC_REROUTING_INFO data structure dereferenced from the extevtdatap in the METAEVENT structure.

Both the rerouting_num (type char *) and the rerouting_addr (type GCLIB_ADDRESS_BLK) fields of the GC_REROUTING_INFO structure contain the same rerouting address string that was explicitly signaled from party A in SIP call transfers or H.450.2 blind call transfers, or from party C via **gc_AcceptInitXfer( )** in H.450.2 supervised call transfers. The rerouting number to be used in the subsequent **gc_MakeCall( )** at party B can be copied from either element, but must not be a concatenation of both elements because they each contain the same character string.

The remaining elements of the GCLIB_ADDRESS_BLK structure dereferenced from rerouting_addr contain the following:

address_type
    GCADDRTYPE_IP

address_plan
    GCADDRPLAN_UNKNOWN

sub_address
    0 (unused)

sub_address_type
    0 (unused)

sub_address_plan
    0 (unused)

Variance for SIP

When party B (the Transferred party) accepts a transfer request via **gc_AcceptXfer( )** no notification is sent to party A (the Transferor or Transferring party). No message is sent to party A until the accepted transfer succeeds or fails.

Variance for SIP

When party B (Transferee or Transferred party) accepts a transfer request via **gc_AcceptXfer( )**, a 202 Accepted message is sent to party A (the Transferor or Transferring party). The call control library at party A may optionally generate a GCEV_INVOKE_XFER_ACCEPTED event to notify the application of the acceptance if that event has been enabled for that line device with **gc_SetConfigData( )**.

Note: Compliance with RFC3515 requires that a NOTIFY(100 Trying) message with Subscription-State=Active be sent in addition to 202 Accepted when a REFER transfer request is accepted. That NOTIFY message (and the required response) not currently supported by the Global Call IP Call Control Library.

7.3.4 gc_AnswerCall( ) Variances for IP

This function is only supported in asynchronous mode.

The **rings** parameter is ignored.

Coders can be set in advance of using **gc_AnswerCall( )** by using **gc_SetUserInfo( )**. See Section 7.3.25, "gc_SetUserInfo( ) Variances for IP", on page 337 for more information.

The following code example shows how to use the **gc_SetUserInfo( )** function to set coder information before calls are answered using **gc_AnswerCall( )**.

```
/* Specifying coders before answering calls */
LINEDEV ldev;
CRN crn;
GC_PARM_BLK *target_datap;
/* Define Coder */
IP_CAPABILITY a_DefaultCapability;
gc_OpenEx(&ldev, ":N_iptB1T1:M_ipmB1C1:P_H323", EV_ASYNC, 0);

/* wait for GCEV_OPENEX event ... */

/* Set default coder for this ldev */
target_datap = NULL;
memset(&a_DefaultCapability,0,sizeof(IP_CAPABILITY));
a_DefaultCapability.capability = GCCAP_AUDIO_g7231_5_3k;
a_DefaultCapability.direction = IP_CAP_DIR_LCLTRANSMIT;
a_DefaultCapability.type = GCCAPTYPE_AUDIO;
a_DefaultCapability.extra.audio.frames_per_pkt = 1;
a_DefaultCapability.extra.audio.VAD = GCPV_DISABLE;
gc_util_insert_parm_ref(&target_datap, GCSET_CHAN_CAPABILITY,
IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY),
&a_DefaultCapability);
```

```
/* set both receive and transmit coders to be the same (since
   the IPTxxx board does not support asymmetrical coders */
memset(&a_DefaultCapability,0,sizeof(IP_CAPABILITY));
a_DefaultCapability.capability = GCCAP_AUDIO_g7231_5_3k;
a_DefaultCapability.direction = IP_CAP_DIR_LCLRECEIVE;
a_DefaultCapability.type = GCCAPTYPE_AUDIO;
a_DefaultCapability.extra.audio.frames_per_pkt = 1;
a_DefaultCapability.extra.audio.VAD = GCPV_DISABLE;
gc_util_insert_parm_ref(&target_datap, GCSET_CHAN_CAPABILITY,
IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY),
&a_DefaultCapability);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, target_datap, GC_ALLCALLS);
gc_util_delete_parm_blk(target_datap);
gc_WaitCall(ldev, NULL, NULL, 0, EV_ASYNC);

/*... Receive GCEV_OFFERED ... */

/*... Retrieve crn from metaevent... */

gc_AnswerCall(crn, 0, EV_ASYNC);

/*... Receive GCEV_ANSWERED ... */
```

Variance for H.323

The **gc_AnswerCall( )** function is used to send the Q.931 CONNECT message to the originating endpoint.

Variance for SIP

The **gc_AnswerCall( )** function is used to send the 200 OK message to the originating endpoint.

7.3.5 gc_CallAck( ) Variances for IP

This function is only supported in asynchronous mode.

The **callack_blkp** parameter must be a pointer to a GC_CALLACK_BLK structure that contains a type field with a value of GCACK_SERVICE_PROC. The following code example shows how to set up a GC_CALLACK_BLK structure and issue the **gc_CallAck( )** function.

```
GC_CALLACK_BLK gcCallAckBlk;
memset(&gcCallAckBlk, 0, sizeof(GC_CALLACK_BLK));
gcCallAckBlk.type = GCACK_SERVICE_PROC;
rc = gc_CallAck(crn, &gcCallAckBlk, EV_ASYNC);
```

The application can configure whether the Proceeding message is sent manually using the **gc_CallAck( )** function or whether it is sent automatically by the stack. See Section 4.18, “Configuring the Sending of the Proceeding Message”, on page 206 for more information.

Variance for H.323

The **gc_CallAck( )** function is used to send the Proceeding message to the originating endpoint.

Variance for SIP

The **gc_CallAck( )** function is used to send the 100 Trying message to the originating endpoint.

7.3.6 gc_DropCall( ) Variances for IP

This function is only supported in asynchronous mode.

The **cause** parameter can be any of the generic cause codes documented in the **gc_DropCall( )** function reference page in the *Global Call API Library Reference* or a protocol-specific cause code as described below.

Variance for H.323

Allowable protocol-specific cause codes are prefixed by IPEC_H225 or IPEC_Q931 in Chapter 10, "IP-Specific Event Cause Codes".

Variance for SIP

Allowable protocol-specific cause codes are prefixed by IPEC_SIP in Chapter 10, "IP-Specific Event Cause Codes".

Note: Cause codes and reasons are only supported when **gc_DropCall( )** is issued while the call is in the Offered state.

7.3.7 gc_Extension( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_Extension( )** function can be used for the following purposes:

- retrieving call-related information
- getting notification of underlying protocol connection or disconnection state transitions
- getting notification of media streaming initiation and termination in both the transmit and receive directions
- specifying which DTMF types, when detected, provide notification to the application
- sending DTMF digits
- retrieving protocol messages (Q.931, H.245, and registration)
- sending protocol messages (Q.931, H.245, and registration)
- performing T.38 fax server operations

Table 17 shows the valid extension IDs and their purpose.

Table 17. Valid Extension IDs for the gc_Extension( ) Function

Extension ID	Description
IPEXTID_CHANGEMODE	Used with **gc_Extension( )** for the following T.38 fax server operations: • initiating a switch from an audio session to a T.38 fax session • initiating a switch from a T.38 fax session to an audio session • accepting a request to switch from audio to T.38 fax or vice versa • rejecting a request to switch from audio to T.38 fax or vice versa Also used in GCEV_EXTENSION events to provide notification of incoming messages including: • a RequestMode (H.323) or REINVITE (SIP) message indicating a request to switch from audio to T.38 fax • a RequestMode (H.323) or REINVITE (SIP) message indicating a request to switch from T.38 fax to audio • a RequestModeAck (H.323) or 200 OK (SIP) message indicating that a switch to audio or T.38 fax has completed successfully See Section 4.26, "T.38 Fax Server", on page 247 for more information.
IPEXTID_FOIP	Used in GCEV_EXTENSION events for notification of information related to fax. See Section 4.17, "Enabling and Disabling Unsolicited Notification Events", on page 205 for more information.
IPEXTID_GETINFO	Used to retrieve call-related information. See Section 4.4, "Retrieving Current Call-Related Information", on page 115 for more information.
IPEXTID_IPPROTOCOL_STATE	Used in GCEV_EXTENSION events for notification of intermediate protocol states, such as, Q.931 and H.245 session connections and disconnections. See Section 4.17, "Enabling and Disabling Unsolicited Notification Events", on page 205 for more information.
IPEXTID_MEDIAINFO	Used in GCEV_EXTENSION events for notification of the initiation and termination of media streaming in the transmit and receive directions. In the case of media streaming connection notification, the datatype of the parameter is IP_CAPABILITY and consists of the coder configuration that resulted from the capability exchange with the remote peer. See Section 4.17, "Enabling and Disabling Unsolicited Notification Events", on page 205 for more information.
IPEXTID_MSGINFO	Used in GCEV_EXTENSION events for receiving SIP messages with MIME-encoded information in the message body. See Section 4.7, "Using MIME Bodies in SIP Messages (SIP-T)", on page 146, for more information. The supported parameter sets are: • IPSET_MIME • IPSET_MIME_200OK_TO_BYE
IPEXTID_RECEIVE_DTMF	Used to select which DTMF types, when detected, provide notification to the application. See Section 4.17, "Enabling and Disabling Unsolicited Notification Events", on page 205 for more information.
IPEXTID_RECEIVEMSG	Used in GCEV_EXTENSION events when SIP, Q.931, H.245, and non-standard registration messages are received.

Table 17. Valid Extension IDs for the gc_Extension( ) Function

Extension ID	Description
IPEXTID_SEND_DTMF	Used to send DTMF digits. When this call is successful, the sending side receives a GCEV_EXTENSIONCMPLT event with the same ext_id. The remote side receives a GCEV_EXTENSION event with IPEXTID_RECEIV_DTMF but only when configured for notification of a specific type of DTMF. See Section 4.17, “Enabling and Disabling Unsolicited Notification Events”, on page 205 for more information.
IPEXTID_SENDMSG	Used to send SIP, H.245, Q.931, and RAS messages. When using this Extension ID, the first parameter inserted into the GC_PARM_BLK must be from one of the following parameter sets: • IPSET_MSG_H245 • IPSET_MSG_Q931 • IPSET_MSG_REGISTRATION • IPSET_MSG_SIP • IPSET_PROTOCOL When the **gc_Extension( )** function completes successfully, the sending side receives a GCEV_EXTENSIONCMPLT event with the same ext_id. The remote side receives a GCEV_EXTENSION event with an ext_id field value of IPEXTID_RECEIVEMSG.

The **gc_Extension**( ) function is only used in the context of a call where the protocol is already known, therefore the protocol does not need to be specified. When protocol-specific information is specified and it is not of the correct protocol type, for example, attempting to send a Q.931 FACILITY message in a SIP call, the operation fails.

See the Section 4.4.2, “Examples of Retrieving Call-Related Information”, on page 119 for a code example showing how to identify the type of extension event and extract the related information.

7.3.8 gc_GetAlarmParm( ) Variances for IP

The **gc_GetAlarmParm**( ) function can be used to get QoS threshold values. The function parameter values in this context are:

linedev

The media device handle, retrieved using the **gc_GetResourceH**( ) function. See Section 4.22.2, “Retrieving the Media Device Handle”, on page 215 for more information.

aso_id

The alarm source object ID. Set to ALARM_SOURCE_ID_NETWORK_ID.

ParmSetID

Must be set to ParmSetID_qosthreshold_alarm.

alarm_parm_list

A pointer to an ALARM_PARM_FIELD structure. The alarm_parm_number field is not used. The alarm_parm_data field is of type GC_PARM, which is a union. In this context, the type used is void *pstruct, and is cast as a pointer to an IPM_QOS_THRESHOLD_INFO structure, which includes an IPM_QOS_THRESHOLD_DATA structure that contains the parameters representing threshold values. See the IPM_QOS_THRESHOLD_INFO structure in the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information. The thresholds supported by Global Call for HMP are QOSTYPE_LOSTPACKETS, QOSTYPE_JITTER, QOSTYPE_RTCPTIMEOUT, and QOSTYPE_RTPTIMEOUT.

mode
Must be set to EV_SYNC.

Note: Applications **must** include the *gcipmlib.h* header file before Global Call can be used to set or retrieve QoS threshold values.

See Section 4.22.3, "Setting QoS Threshold Values", on page 215 for code examples.

7.3.9 gc_GetCallInfo( ) Variances for IP

The **gc_GetCallInfo( )** function can be used to retrieve calling (ANI) or called party (DNIS) information such as an IP address, an e-mail address, an E.164 number, a URL, or the call identifier (Call ID) used by the underlying protocol to globally, uniquely identify the call. The values of the **info_id** parameter that are supported for both SIP and H.323 are:

ORIGINATION_ADDRESS
the calling party information (equivalent to ANI)

DESTINATION_ADDRESS
the called party information (equivalent to DNIS)

IP_CALLID
the globally unique identifier used by the underlying protocol to identify the call (Call ID or GUID)

Two additional, SIP-specific values for the **info_id** parameter that allow retrieval of information from the From URI and To URI SIP message fields are described below under the "Variance for SIP" heading.

When an **info_id** of ORIGINATION_ADDRESS (ANI) is specified and the function completes successfully, the **valuep** string is a concatenation of values delimited by a pre-determined character (configurable in the IPCCLIB_START_DATA data structure used by **gc_Start( )**; the default is a comma).

When an **info_id** of DESTINATION_ADDRESS (DNIS) is specified and the function completes successfully, the **valuep** string is a concatenation of values delimited by a pre-determined character (configurable in the IPCCLIB_START_DATA data structure used by **gc_Start( )**; the default is a comma). The IP address of the destination gateway (that is processing the DNIS) is **not** included in the string.

When an **info_id** of IP_CALLID (Call ID) is specified and the function completes successfully, the buffer pointed to by the **valuep** argument contains the globally unique identifier used by the underlying protocol to identify the call. The size and datatype of the Call ID depends on the protocol. To assure adequate buffer size when the protocol is unknown, use the IP_CALLIDSIZE define to allocate a buffer that is large enough to hold any type of Call ID value (i.e., either an H.323 array of octets or a SIP string).

Note: For outbound calls the **gc_GetCallInfo( )** function can be used to retrieve valid Call ID information only after the Proceeding state.

The **gc_GetCallInfo( )** function can also be used to query the protocol used by a call. The **info_id** parameter should be set to CALLPROTOCOL and the **valuep** parameter returns a pointer to an integer that is one of the following values:

- CALLPROTOCOL_H323
- CALLPROTOCOL_SIP

Note: For an inbound call, the **gc_GetCallInfo( )** function can be used to determine the protocol any time after the GCEV_OFFERED event is received and before the GCEV_DISCONNECTED event is received.

Variance for H.323

When retrieving calling (ANI) information, the following rules apply. Any section in the string that includes a prefix (TA:, TEL:, or NAME:) has been inserted as an alias by the originating party. Any section in the string that does not include a prefix has been inserted as a **calling party** number (Q.931) by the originating party.

When retrieving called party (DNIS) information, the following rules apply. Any section in the string that includes a prefix (TA:, TEL:, or NAME:) has been inserted as an alias by the originating party. Any section in the string that does not include a prefix has been inserted as a **called party** number (Q.931) by the originating party.

When retrieving Call ID information, the buffer pointed to by the **valuep** argument contains an array of octets. The size of this array is IP_H323_CALLIDSIZE bytes. To assure adequate buffer size when the protocol is unknown, use the IP_CALLIDSIZE define to create a buffer that is large enough to hold any type of Call ID value (i.e., either H.323 or SIP).

Variance for SIP

When retrieving calling party (ANI) or called party (DNIS) information, prefixes (such as TA:, TEL:, and NAME:) are **not** used.

When retrieving calling party (ANI) information, the address is taken from the SIP From: header, and is accessible in one of two forms by using one of the following parameter IDs in the function call:

ORIGINATION_ADDRESS

Returns the simple origination address in the form

alice@192.168.1.10

ORIGINATION_ADDRESS_SIP

Returns a SIP-specific origination address that includes additional From URI parameters and tags. The format used is

sip: alice@192.168.1.10;tag=0-13c4-4059c361-23d07406-72fe

When retrieving called party (DNIS) information, the address is taken from the SIP To: header, and is accessible in one of two forms by using one of the following parameter IDs in the function call:

DESTINATION_ADDRESS
Returns the simple destination address in the form
  user@127.0.0.1

DESTINATION_ADDRESS_SIP
Returns a SIP-specific destination address that includes additional To URI parameters in the form
  sip: userB@127.0.0.1;user=Steve

When retrieving Call ID information, the buffer pointed to by the **valuep** argument contains a NULL-terminated string. The maximum size of this string is IP_SIP_CALLIDSIZE bytes. To assure adequate buffer size when the protocol is unknown, use the IP_CALLIDSIZE define. This will assure the buffer is large enough to hold any type of Call ID value (i.e., either H.323 or SIP).

Retrieving SIP Call ID via gc_GetCallInfo( )

The following code example illustrates retrieval of the SIP Call ID using a **gc_GetCallInfo( )** call.

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_iptB1T1:P_SIP, :N_iptB1T2:P_SIP, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been called
 *    to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>

/*
 * Assume the 'crn' parameter holds the CRN associated with the detected GCEV_OFFERED event.
 */

int print_call_info(CRN crn)
{
    GC_INFO gc_error_info;             /* GlobalCall error information data */
    char cid_buff[IP_SIP_CALLIDSIZE];  /* buffer large enough to hold SIP Call-ID value */

    if(gc_GetCallInfo(crn, IP_CALLID, cid_buff) != GC_SUCCESS)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetCallInfo(IP_CALLID) on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,"\
                " CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg, gc_error_info.ccLibId,
                gc_error_info.ccLibName, gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    printf ("gc_GetCallInfo(IP_CALLID) on crn: 0x%lx, returned - %s\n", crn, cid_buff);

    return(0);
}
```

7.3.10 gc_GetCTInfo( ) Variances for IP

The **gc_GetCTInfo( )** function can be used to retrieve product information (via the CT_DEVINFO structure) for the media sub-device (ipm) attached to the network device (ipt). If no media device is associated with the network device, the function returns as though not supported.

7.3.11 gc_GetResourceH( ) Variances for IP

The **gc_GetResourceH**( ) function can be used to retrieve the media device (ipm device) handle, which is required by GCAMS functions, such as, **gc_SetAlarmParm**( ) and **gc_GetAlarmParm**( ) to set and retrieve QoS threshold values. The function parameter values in this context are:

linedev
: the network device, that is, the Global Call line device retrieved by the **gc_OpenEx**( ) function

resourcehp
: the address where the media device handle is stored when the function completes

resourcetype
: GC_MEDIADEVICE

Note: Applications **must** include the *gcipmlib.h* header file before Global Call can be used to set or retrieve QoS threshold values.

The other resource types including GC_NETWORKDEVICE (for a network device), GC_VOICEDEVICE (for a voice device), and GC_NET_GCLINEDEVICE (to retrieve the Global Call line device handle when the media handle is known) are also supported.

Note: The GC_VOICEDEVICE option above applies only if the voice device was opened with the line device or opened separately and subsequently attached to the line device.

7.3.12 gc_GetXmitSlot( ) Variances for IP

The **gc_GetXmitSlot**( ) function can be used to get the transmit time slot information for an IP Media device. The function parameter values in this context are:

linedev
: The Global Call line device handle for an IP device (that is, the handle returned by **gc_OpenEx**( ) for a device with :N_iptBxTy in the **devicename** parameter and a media device attached).

sctsinfop
: A pointer to the transmit time slot information for the IP Media device (a pointer to a CT Bus time slot information structure).

7.3.13 gc_InitXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

The **parmblkp** and **ret_rerouting_infopp** parameters are ignored and should be set to NULL. The **gc_InitXfer( )** function returns -1 if invalid parameter are specified.

Variance for H.323 (H.450.2)

The **gc_InitXfer( )** function has an associated GCEV_INIT_XFER termination event that is received on the specified CRN. This termination event indicates that the initiate transfer request was successful and that party C has sent a positive acknowledgement.

Variance for SIP

The **gc_InitXfer( )** function does not cause any SIP message to be sent to either of the remote parties, and is used only for purposes of synchronizing the Global Call state machine. The GCEV_INIT_XFER termination event that the Transferor receives on the specified CRN after calling **gc_InitXfer( )** is a "dummy" event whose only purpose is to allow synchronization of the Global Call state machine.

7.3.14 gc_InvokeXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

Variance for H.323 (H.450.2)

The party A application is notified by GCEV_INVOKE_XFER_REJ if the remote party receiving the call transfer request rejects the request, or by GCEV_INVOKE_XFER_FAIL if the request fails for some reason, but there is **no** notification if the request is accepted. The only notification party A receives in a successful transfer is the GCEV_INVOKE_XFER event, which does not necessarily mean that the transferred call between party B and party C was connected, only that it was confirmed to be delivered. Specifically, it indicates that ALERTING or CONNECT was received from party C on the transferred call.

Table 18 identifies the protocol-specific variances in parameters for **gc_InvokeXfer( )**.

Table 18. gc_InvokeXfer( ) Supported Parameters for H.450.2

Parameter	Meaning
crn	For all transfers, CRN of primary call.
extracrn	For a supervised call transfer, parameter value must be the CRN of the secondary/consultation call with party C. For blind call transfers, parameter value must be zero.

Table 18. gc_InvokeXfer( ) Supported Parameters for H.450.2 (Continued)

Parameter	Meaning
numberstr	Ignored in supervised call transfer – set to NULL. For blind call transfer, used to provide address of party C (the rerouting address) as a string. Signaled to party B in the GCEV_REQ_XFER event. Format can be: • transport address, for example, “TA:146.152.0.1” • E.164 alias, for example, “TEL:9739674700” • host address, for example, “NAME: myhostname” **Note:** When using the GC_MAKECALL_BLK ***makecallp** parameter to specify the rerouting address via a data structure, this parameter must be set to NULL.
makecallp	Ignored in supervised call transfer – set to NULL. For blind call transfer, used to provide address of party C (the rerouting address) in a GC_MAKECALL_BLK data structure. Signaled to party B in the GCEV_REQ_XFER event. **Note:** When using the char ***numberstr** parameter to specify the rerouting address as a string, this parameter must be set to NULL.
timeout	Ignored. H.450.2 timers (T1, T2, T3, T4) are implicitly maintained at 20 seconds) – set to NULL.

Table 19 through Table 22 list the possible event failure cause values.

Table 19. H.450.2 CtInitiate Errors Received from the Network

ctInitiate Error	Result Values	GC Event
notAvailable	CC: IPEC_H450NotAvailable GC: GCRV_REMOTEREJ_UNAVAIL	GCEV_INVOKE_XFER_REJ
invalidCallState	CC: IPEC_H450InvalidCallState GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_FAIL
invalidReroutingNumber	CC: IPEC_H4502InvalidReroutingNumber GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_REJ
unrecognizedCallIdentity	CC: IPEC_H4502UnrecognizedCallIdentity GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_FAIL
establishmentFailure	CC: IPEC_H4502EstablishmentFailure GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_FAIL
supplementaryServiceInteractionNotAllowed	CC: IPEC_H450SuppServInteractionNotAllowed GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_REJ
unspecified	CC: IPEC_H4502Unspecified GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_REJ

Table 20. H.450.2 CtIdentify Errors Received From the Network

ctIdentify Error	Result Values	GC Event
notAvailable	CC: IPEC_H450TRTSENotAvailable GC: GCRV_REMOTEREJ_UNAVAIL	GCEV_INVOKE_XFER_REJ
invalidCallState	CC: IPEC_H450TRTSEInvalidCallState GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_FAIL

Table 20. H.450.2 CtIdentify Errors Received From the Network (Continued)

ctIdentify Error	Result Values	GC Event
supplementaryServiceInteractionNotAllowed	CC: IPEC_H450TRTSESuppServInteractionNotAllowed GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_REJ
unspecified	CC: IPECH4502TRTSEUnspecified GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_REJ

Table 21. H.450.2 CtSetup Errors Received From the Network

ctSetup Error	Result Values	GC Event
notAvailable	CC: IPEC_H450NotAvailable GC: GCRV_REMOTEREJ_UNAVAIL	GCEV_INVOKE_XFER_REJ
invalidCallState	CC: IPEC_H450InvalidCallState GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_FAIL
invalidReroutingNumber	CC: IPEC_H4502InvalidReroutingNumber GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_REJ
unrecognizedCallIdentity	CC: IPEC_H4502UnrecognizedCallIdentity GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_FAIL
supplementaryServiceInteractionNotAllowed	CC: IPEC_H450SuppServInteractionNotAllowed GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_REJ
unspecified	CC: IPEC_H4502Unspecified GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_REJ

Table 22. H.450.2 CT Timer Expiry

Endpoint – Timer	Result Values	GC Event
TRGSE – T1	CC: IPEC_H450CTT1Timeout GC: GCRV_TIMEOUT	GCEV_INVOKE_XFER_FAIL
TRGSE – T3	CC: IPEC_H450CTT3Timeout GC: GCRV_TIMEOUT	GCEV_INVOKE_XFER_FAIL

Variance for SIP

The application at party A may optionally be notified by a GCEV_INVOKE_XFER_ACCEPTED event that the transfer request has been accepted by the remote party to which it was sent. (This event has no equivalent in H.450.2.) This event is optional, and is disabled by default. The event may be enabled and disabled on a per-line-device basis via the **gc_SetConfigData( )** function as shown in the following code example.

```
//enable GCEV_INVOKE_XFER_ACCEPTED event for SIP call transfer
GC_PARM_BLK *t_pParmBlk = NULL;
long        request_id;

gc_util_insert_parm_val(&t_parmBlkl, GCSET_CALLEVENT_MSK, GCACT_ADDMSK,
                        sizeof(long), GCMSK_INVOKE_XFER_ACCEPTED);
```

```
gc_SetConfigData(GCTGT_GCLIB_CHAN,ldev,t_pParmBlk,0,GCUPDATE_IMMEDIATE,&request_id,EV_SYNC);

gc_util_delete_parm_blk(t_pParmBlk)
```

The specific meaning of the GCEV_INVOKE_XFER termination event for successful transfers is dependant on the application and the transfer scenario(s) it uses. The possible outcomes when Global Call is used by all parties include the following:

- If party A drops the primary call in unattended transfers before the transfer completes, party A does not receive any GCEV_INVOKE_XFER event at all.
- If party B drops the primary call in unattended transfers before the transfer completes, party A receives a GCEV_INVOKE_XFER event that only signifies that party B has sent INVITE to party C.
- For attended transfers or unattended transfers where the primary call is maintained during the transfer, party A receives a GCEV_INVOKE_XFER event which indicates that the transferred call was actually connected between party B and party C.

Table 23 identifies the protocol-specific variances in parameters for **gc_InvokeXfer( )**.

Table 23. gc_InvokeXfer( ) Supported Parameters for SIP

Parameter	Meaning
crn	The CRN of the call between party A and the remote party receiving the transfer request. This is the primary call in an unattended (blind) call transfer, but may be either call for an attended (supervised) transfer.
extracrn	For an attended (supervised) call transfer, the CRN of the call between party A and the remote party *not* receiving the transfer request (i.e. the call not specified in the **crn** parameter). For unattended (blind) call transfers, must be zero.
numberstr	For attended (supervised) call transfers, this parameter is ignored. Set to NULL. For an unattended (blind) call transfer, the address of party C (the rerouting address, which will be signaled to party B) as a string. This address is of the form `user@host; param=value` where • `user` is a user name or phone number • `host` is a domain name or IP address • `param=value` is an optional additional parameter For additional information on rules for destination addresses, see Section 7.3.16.3, “Forming a Destination Address String”, on page 315 under the “Variance for SIP” heading. **Note:** When using the GC_MAKECALL_BLK ***makecallp** parameter to specify the rerouting address, this parameter must be set to NULL.
makecallp	For attended (supervised) call transfers, this parameter is Ignored. Set to NULL. For an unattended (blind) call transfer, the address of party C (the rerouting address, which will be signaled to party B) as a GC_MAKECALL_BLK data structure. **Note:** When using the char ***numberstr** parameter to specify the rerouting address, this parameter must be set to NULL.
timeout	Ignored. Set to NULL.

The application may optionally set the specific information in the header fields of the SIP REFER message that is sent by this function by configuring a GC_PARM_BLK before calling **gc_InvokeXfer( )**, as described in Section 4.6, “Setting and Retrieving SIP Message Header

Fields”, on page 130. Table 24 lists the header fields that can be set in REFER messages and the corresponding parameter IDs along with examples of field values.

Table 24. SIP Header Fields Settable in REFER Messages

Field Name	GC Parameter ID (Set ID: IPSET_SIP_MSGINFO)	Example Field Value
Request URI	IPPARM_REQUEST_URI	146.152.212.67:5060
From	IPPARM_FROM	From: Transferor <sip:146.152.212.43>;tag=0-13c4-408c7921-1026900f-ed5;myname
To	IPPARM_TO	To: Transferee <sip:146.152.212.67:5060>;tag=0-13c4-408c7921-10268fdd-6a19
From Display	IPPARM_FROM_DISPLAY	Transferor
To Display	IPPARM_TO_DISPLAY	Transferee
Call ID	IPPARM_CALLID_HDR	48cabd0-0-13c4-408c7921-10268fdd-1563@146.152.212.67
Contact URI	IPPARM_CONTACT_URI	sip:146.152.212.43
Contact Display	IPPARM_CONTACT_DISPLAY	Transferor
Referred-By	IPPARM_REFERRED_BY	Referred-By: <sip:146.152.212.43>
Replaces	IPPARM_REPLACES	Replaces: 48cae78-0-13c4-408c7923-1026947b-1078@146.152.212. 67;to-tag=0-13c4-408c7923-102694a3-6942;from-tag=0-13c4-408c7923-1026947b-7b6

7.3.15 gc_Listen( ) Variances for IP

The **gc_Listen**( ) function is supported in both synchronous and asynchronous modes. The function is blocking in synchronous mode.

Note: For line devices that comprise media (ipm) and voice (dxxx) devices, routing is only done on the media devices. Routing of the voice devices must be done using the Voice API (dx_ functions).

7.3.16 gc_MakeCall( ) Variances for IP

This function is only supported in asynchronous mode.

Global Call supports multiple IP protocols on a single IPT Network device. See Section 2.3.3, “IPT Network Devices”, on page 44 for more information. When using a multi-protocol network device (that is, one opened in P_IP mode), the application specifies the protocol in the associated GC_MAKECALL_BLK structure, using the set ID IPSET_PROTOCOL, the parameter ID IPPARM_PROTOCOL_BITMASK, and one of the following values:

- IP_PROTOCOL_SIP
- IP_PROTOCOL_H323

A network device that is opened in multi-protocol mode defaults to IP_PROTOCOL_H323 if the protocol is not explicitly set in the makecall block.

Note: Applications should **not** use the **gc_SetUserInfo( )** function to set the IP protocol.

When making calls on devices that support only one protocol, it is not necessary to include an IPSET_PROTOCOL element in the makecall block. If the application tries to include an IPSET_PROTOCOL element in the makecall block that conflicts with the protocol supported by the device, the application receives an error.

When using SIP, if the remote site does not respond to an outgoing INVITE sent by the call control library, the **gc_MakeCall( )** function times out after 32 seconds and generates a GCEV_DISCONNECTED event. In this no-response scenario, if the application attempts to drop the call before the 32 second timeout is reached, the library sends a CANCEL to the remote site. If there is no response by the remote site to the CANCEL, there will be an additional 32 second timeout, at the end of which a GCEV_DISCONNECTED event will be reported.

7.3.16.1 Configurable Call Parameters

Call parameters can be specified when using the **gc_MakeCall( )** function. The parameters values specified are only valid for the duration of the current call. At the end of the current call, the default parameter values for the specific line device override these parameter values. The **makecallp** parameter of the **gc_MakeCall( )** function is a pointer to the GC_MAKECALL_BLK structure. The GC_MAKECALL_BLK structure has a gclib field that points to a GCLIB_MAKECALL_BLK structure. The ext_datap field within the GCLIB_MAKECALL_BLK structure points to a GC_PARM_BLK structure with a list of the parameters to be set as call values. The parameters that can be specified through the ext_datap pointer depend on the protocol used, H.323 or SIP and are described in the subsections following.

Variance for H.323

Table 25 shows the call parameters that can be specified when using **gc_MakeCall( )** with H.323.

Table 25. Configurable Call Parameters When Using H.323

Set ID	Parameter ID(s) and Data Types
GCSET_CHAN_CAPABILITY	IPPARM_LOCAL_CAPABILITY Datatype IP_CAPABILITY. See the reference page for IP_CAPABILITY on page 385 for more information. **Note:** If no transmit/receive coder type is specified, any supported coder type is accepted.
IPSET_CALLINFO See Section 8.2.2, "IPSET_CALLINFO", on page 358 for more information.	IPPARM_CONNECTIONMETHOD Enumeration, with one of the following values: • IPPARM_CONNECTIONMETHOD_FASTSTART • IPPARM_CONNECTIONMETHOD_SLOWSTART See Section 4.2, "Using Fast Start and Slow Start Setup", on page 104 for more information.
Notes: The term "String" implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).	

Table 25. Configurable Call Parameters When Using H.323 (Continued)

Set ID	Parameter ID(s) and Data Types
IPSET_CALLINFO (continued)	IPPARM_CALLID Array of octets, length = MAX_IP_H323_CALLID_LENGTH
	IPPARM_DISPLAY String, max. length = MAX_DISPLAY_LENGTH (82), null-terminated
	IPPARM_H245TUNNELING Enumeration, with one of the following values: • IP_H245TUNNELING_ON or IP_H245TUNNELING_OFF See Section 4.19, “Enabling and Disabling H.245 Tunneling”, on page 206 for more information.
	IPPARM_PHONELIST String, max. length = 131.
	IPPARM_USERUSER_INFO String, max. length = MAX_USERUSER_INFO_LENGTH (131 bytes)
IPSET_CONFERENCE	IPPARM_CONFERENCE_GOAL Enumeration with one of the following values: • IP_CONFERENCEGOAL_UNDEFINED • IP_CONFERENCEGOAL_CREATE • IP_CONFERENCEGOAL_JOIN • IP_CONFERENCEGOAL_INVITE • IP_CONFERENCEGOAL_CAP_NEGOTIATION • IP_CONFERENCEGOAL_SUPPLEMENTARY_SRVC
IPSET_NONSTANDARDDATA See Section 8.2.18, “IPSET_NONSTANDARDDATA”, on page 368 for more information.	Either: • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_NONSTANDARDDATA_OBJID Unsigned Int[ ], max. length =MAX_NS_PARM_OBJID_LENGTH (40) or • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_H221NONSTANDARD Datatype IP_H221NONSTANDARD

Notes:
The term “String” implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).

Table 25. Configurable Call Parameters When Using H.323 (Continued)

Set ID	Parameter ID(s) and Data Types
IPSET_NONSTANDARDCONTROL See Section 8.2.17, "IPSET_NONSTANDARDCONTROL", on page 368 for more information.	Either: • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_NONSTANDARDDATA_OBJID Unsigned Int[ ], max. length = MAX_NS_PARM_OBJID_LENGTH (40) or • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_H221NONSTANDARD Datatype IP_H221NONSTANDARD
Notes: The term "String" implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).	

Variance for SIP

Table 26 shows the call parameters that can be specified when using **gc_MakeCall( )** with SIP.

Table 26. Configurable Call Parameters When Using SIP

Set ID	Parameter ID and Datatype
GCSET_CHAN_CAPABILITY	IPPARM_LOCAL_CAPABILITY Datatype IP_CAPABILITY. See reference page for IP_CAPABILITY on page 385 for more information. **Note:** If no transmit/receive coder type is specified, any supported coder type is accepted.
IPSET_CALLINFO See Section 8.2.2, "IPSET_CALLINFO", on page 358 for more information.	IPPARM_CONNECTIONMETHOD Enumeration, with one of the following values: • IPPARM_CONNECTIONMETHOD_FASTSTART • IPPARM_CONNECTIONMETHOD_SLOWSTART See Section 4.2, "Using Fast Start and Slow Start Setup", on page 104 for more information.
	IPPARM_CALLID String, max. length = MAX_IP_SIP_CALLID_LENGTH **Note:** Directly manipulating the SIP Call ID message header via IPSET_SIP_MSGINFO and IPPARM_CALLID_HDR will override any value provided here.
Notes: The term "String" implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination). The parameter names used are more closely aligned with H.323 terminology. Corresponding SIP terminology is described in http://www.ietf.org/rfc/rfc3261.txt?number=3261.	

Table 26. Configurable Call Parameters When Using SIP (Continued)

Set ID	Parameter ID and Datatype
IPSET_CALLINFO (continued)	IPPARM_DISPLAY String, max. length = MAX_DISPLAY_LENGTH (82), null-terminated
	IPPARM_PHONELIST String, max. length = 131

Notes:
The term “String” implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).
The parameter names used are more closely aligned with H.323 terminology. Corresponding SIP terminology is described in http://www.ietf.org/rfc/rfc3261.txt?number=3261.

7.3.16.2 Origination Address Information

The origination address can be specified in the origination field of type GCLIB_ADDRESS_BLK in the GCLIB_MAKECALL_BLK structure. The address field in the GCLIB_ADDRESS_BLK contains the actual address and the address_type field in the GCLIB_ADDRESS_BLK structure defines the type (IP address, name, telephone number) in the address field.

Note: The total length of the address string is limited by the value MAX_ADDRESS_LEN (defined in *gclib.h*).

The origination address can be set using the **gc_SetCallingNum( )** function, which is a deprecated function. The preferred equivalent is **gc_SetConfigData( )**. See the *Global Call API Library Reference* for more information.

7.3.16.3 Forming a Destination Address String

Variance for H.323

The destination address is formed by concatenating values from three different sources:

- the GC_MAKECALL_BLK
- the **numberstr** parameter of **gc_MakeCall( )**
- the phone list

The order or precedence of these elements and the rules for forming a destination address are described below.

Notes: *1.* The following description refers to a delimited string. The delimiter is configurable by setting the value of the delimiter field in the IP_CCLIB_START_DATA structure used by the **gc_Start( )** function.

2. The total length of the address string is limited by the value MAX_ADDRESS_LEN (defined in *gclib.h*).

3. The destination address must be a valid address that can be translated by the remote node.

The destination information string is delimited concatenation of the following strings in the order of precedence shown:

1. A string constructed from the destination field of type GCLIB_ADDRESS_BLK in the GCLIB_MAKECALL_BLK. When specifying the destination information in the GCLIB_ADDRESS_BLK, the address field contains the actual address information and the address_type field defines the type (IP address, name, telephone number) in the address. For example, if the address field is "127.0.0.1", the address_type field must be GCADDRTYPE_IP. The supported address types are:
   - GCADDRTYPE_INTL – international telephone number
   - GCADDRTYPE_NAT – national telephone number
   - GCADDRTYPE_LOCAL – local telephone number
   - GCADDRTYPE_DOMAIN – domain name
   - GCADDRTYPE_URL – URL name
   - GCADDRTYPE_EMAIL – e-mail address
2. The **numberstr** parameter in the **gc_MakeCall( )** function. The **numberstr** parameter is treated as a free string that may be a delimited concatenation of more than one section. The application may include a prefix in a section that maps to a corresponding field in the Setup message. See Section 7.3.16.4, "Destination Address Interpretation", on page 318, for more information.
3. Phone list as described in Table 25, "Configurable Call Parameters When Using H.323", on page 312 (and set using IPSET_CALLINFO, IPPARM_PHONELIST). Phone List is treated as a free string that may be a delimited concatenation of more than one section. The application may prefix a section that maps to a corresponding field in the Setup message. See the Section 7.3.16.4, "Destination Address Interpretation", on page 318 for more information.

Variance for SIP

The format of the destination address for a SIP call is:

```
user@host; param=value
```

with the elements representing:

user
: A user name or phone number

host
: A domain name or an IP address

param=value
: An optional additional parameter

When making a SIP call, the destination address is formed according to the following rules in the order of precedence shown:

1. If Phone List (as described in Table 26, "Configurable Call Parameters When Using SIP", on page 314 and identified by IPSET_CALLINFO, IPPARM_PHONELIST) exists, it is taken to construct the global destination-address-string.
2. If the destination address field (of type GCLIB_ADDRESS_BLK in GCLIB_MAKECALL_BLK) exists, it is taken to construct the global destination-address-string. The address_type in GCLIB_ADDRESS_BLK is ignored. If the global destination-

address-string is not empty before setting the parameter, an “@” delimiter is used to separate the two parts.

3. If the **numberstr** parameter from the **gc_MakeCall( )** function exists, it is taken to destination-address-string. If the global destination-address-string is not empty before setting the parameter, a “;” delimiter is used to separate the two parts.

Note: To observe the logic described above, the application may use only one of the APIs to send a string that is a valid SIP address.

The following code examples demonstrate the recommended ways of forming the destination string when making a SIP call. Prerequisite code for setting up the GC_MAKECALL_BLK in all the scenarios described in this section is as follows:

```
GC_MAKECALL_BLK gcmkbl;
GCLIB_MAKECALL_BLK gclib_mkbl = {0};
gcmkbl.cclib = NULL;
gcmkbl.gclib = &gclib_mkbl;
GC_PARM_BLK *target_datap = NULL;

gc_util_insert_parm_val(&target_datap,
                        IPSET_PROTOCOL,
                        IPPARM_PROTOCOL_BITMASK,
                        sizeof(char),
                        IP_PROTOCOL_SIP);
```

Scenario 1 – Making a SIP call to a known IP address, where the complete address (user@host) is specified in the makecall block:

```
char *pDestAddrBlk = "11223344@127.0.0.1";  /* where "11223344" is the
                                             phone number of the user
                                             and "127.0.0.1" is the
                                             IP address of the host */

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

/* calling the function with the MAKECALL_BLK, and numberstr parameter=NULL
   the INVITE dest address will be: 11223344@127.0.0.1 */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout, EV_ASYNC);
```

Scenario 2 – Making a SIP call to a known IP address, where the complete address (user@host) is formed by the combination of the destination address in the makecall block and the phone list element:

```
char *pDestAddrBlk = "127.0.0.1";   /*host*/
char *IpPhoneList = "003227124311"; /*user*/

/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;
```

```
gclib_mkbl.ext_datap = target_datap;

/* calling the function with the MAKECALL_BLK, and numberstr parameter = NULL
   the INVITE dest address will be: 003227124311@127.0.0.1 */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 3 – Making a SIP call to a known IP address, where the complete address (user@host) is formed by the combination of the destination address in the makecall block, a phone list element, and optional parameter (user=phone):

```
char *pDestAddrBlk = "127.0.0.1";  /*host*/
char *IpPhoneList= "003227124311"; /*user*/
char *pDestAddrStr = "user=phone"; /*extra parameter*/

/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK, and numberstr parameter = NULL
   the INVITE dest address will be: 003227124311@127.0.0.1;user=phone */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

7.3.16.4 Destination Address Interpretation

Note: The following information applies when using H.323 only.

Once a destination string is formed as described in the previous section, the H.323 stack treats the string according to the following rules:

- The **first** section of the string is the destination of the next IP entity (for example, a gateway, terminal, the alias for a remote registered entity, etc.) with which the application attempts to negotiate.
- A non-prefixed section in the string is the Q.931 calledPartyNumber and is the **last** section that is processed. Any section following the first non-prefixed section is ignored. Only **one** Q.931 calledPartyNumber is allowed in the destination string.
- One or more prefixed sections (H.225 destinationAddress fields) must appear **before** the non-prefixed section (Q.931 calledPartyNumber).
- When using free strings (**numberstr** parameter or Phone List), if the application wants to prefix buffers, valid buffer prefixes for H.225 addresses are:
  - TA: – IP transport address
  - TEL: – e164 telephone number
  - NAME: – H.323 ID
  - URL: – Universal Resource Locator
  - EMAIL: – e-mail address

The following code examples demonstrate the recommended ways of forming the destination string when making an H.323 call. Prerequisite code for setting up the GC_MAKECALL_BLK in all the scenarios described in this section is as follows:

```
GC_MAKECALL_BLK gcmkbl;
GCLIB_MAKECALL_BLK gclib_mkbl = {0};
gcmkbl.cclib = NULL;
gcmkbl.gclib = &gclib_mkbl;
GC_PARM_BLK *target_datap = NULL;

gc_util_insert_parm_val(&target_datap,
                        IPSET_PROTOCOL,
                        IPPARM_PROTOCOL_BITMASK,
                        sizeof(char),
                        IP_PROTOCOL_H323);
```

Scenario 1 – Making a call to a known IP address, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *pDestAddrStr = "123456";

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK*/
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 2 – Making a call to a known IP address, setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *pDestAddrStr = "TEL:111,TEL:222,76543";

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK*/
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 3 – Making a call to a known IP address, setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *pDestAddrStr = "TEL:111,TEL:222,NAME:myName";
char *IpPhoneList= "003227124311";

/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK*/
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 4 – Making a call to a known IP address, setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *IpPhoneList= "TEL:003227124311,TEL:444,TEL:222,TEL:1234,171717";
/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);
gclib_mkbl.ext_datap = target_datap;

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK, and numberstr
   parameter = NULL */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 5 – While registered, making a call, via the gatekeeper, to a registered entity (using a known H.323 ID), setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = " RegisteredRemoteGW ";  /* The alias of the remote (registered) entity */
char *pDestAddrStr = "TEL:111,TEL:222,987654321";

/* set GCLIB_ADDRESS_BLK with destination string & type (H323-ID) */
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_DOMAIN;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 6 – While registered, making a call, via the gatekeeper, to a registered entity (using a known e-mail address), setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = " user@host.com ";  /* The alias of the remote (registered) entity */
char *pDestAddrStr = "TEL:111,TEL:222,987654321";

/* set GCLIB_ADDRESS_BLK with destination string & type (EMAIL) */
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_EMAIL;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 7 – While registered, making a call, via the gatekeeper, to a registered entity (using a known URL), setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "www.gw1.intel.com";  /* The alias of the remote (registered) entity */
char *pDestAddrStr = "TEL:111,TEL:222,987654321";

/* set GCLIB_ADDRESS_BLK with destination string & type (URL) */
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_URL;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

7.3.16.5 Specifying a Timeout

Note: The following information applies when using H.323 only.

The **timeout** parameter of the **gc_MakeCall( )** function specifies the maximum time in seconds to wait for the establishment of a new call, after receiving the first response to the call. This value corresponds to the **Q.931\connectTimeOut** parameter. If the call is not established during this time, the Disconnect procedure is initiated. The default value is 120 seconds.

In addition to the **Q.931\connectTimeOut** parameter described in Section 7.3.16, "gc_MakeCall( ) Variances for IP", on page 311, two other parameters that affect the timeout behavior, but are not configurable are:

Q931\responseTimeOut
The maximum time in seconds to wait for the first response to a new call. If no response is received during this time, the Disconnect procedure is initiated. The default value is 4 seconds.

h245\timeout:
The maximum time in seconds to wait for the called party to acknowledge receipt of the capabilities it sent. The default value is 40 seconds.

Note: When using the H.323 protocol, the application may receive a timeout when trying to make an outbound call if network congestion is encountered and a TCP connection cannot be established. In this case, the SETUP message is not sent on the network.

7.3.16.6 Code Examples

H.323-Specific Code Example

The following code example shows how to make a call using the H.323 protocol.

```
/* Make an H323 IP call on line device ldev */
void MakeH323IpCall(LINEDEV ldev)
{
   char *IpDisplay = "This is a Display"; /* display data */
   char *IpPhoneList= "003227124311"; /* phone list */
   char *IpUUI = "This is a UUI";    /* user to user information string */
   char *pDestAddrBlk = "127.0.0.1"; /* destination IP address for MAKECALL_BLK*/
   char *pSrcAddrBlk = "987654321";  /* origination address for MAKECALL_BLK*/
   char *pDestAddrStr = "123456";    /* destination string for gc_MakeCall() function*/
   char *IpNSDataData = "This is an NSData data string";
   char *IpNSControlData = "This is an NSControl data string";
   char *IpCommonObjId = "1 22 333 4444"; /* unique format */
   IP_H221NONSTANDARD appH221NonStd;
   appH221NonStd.country_code = 181; /* USA */
   appH221NonStd.extension = 11;
   appH221NonStd.manufacturer_code = 11;
   int ChoiceOfNSData = 1;
   int ChoiceOfNSControl = 1;
   int rc = 0;
   CRN crn;
   GC_MAKECALL_BLK gcmkbl;
   int MakeCallTimeout = 120;

   /* initialize GCLIB_MAKECALL_BLK structure */
   GCLIB_MAKECALL_BLK gclib_mkbl = {0};
```

```
/* set to NULL to retrieve new parameter block from utility function */
GC_PARM_BLK *target_datap = NULL;
gcmkbl.cclib = NULL; /* CCLIB pointer unused */
gcmkbl.gclib = &gclib_mkbl;

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

/* set GCLIB_ADDRESS_BLK with origination string & type*/
strcpy(gcmkbl.gclib->origination.address,pSrcAddrBlk);
gcmkbl.gclib->origination.address_type = GCADDRTYPE_NAT;

/* set signaling PROTOCOL to H323. default is H323 if device is multi-protocol */
rc = gc_util_insert_parm_val(&target_datap,
                             IPSET_PROTOCOL,
                             IPPARM_PROTOCOL_BITMASK,
                             sizeof(char),
                             IP_PROTOCOL_H323);

/* initialize IP_CAPABILITY structure */
IP_CAPABILITY t_Capability = {0};
/* configure a GC_PARM_BLK with four coders, display, phone list and UUI message: */
/* specify and insert first capability parameter data for G.7231 coder */
t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLTRANSMIT;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLRECEIVE;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert second capability parameter data for G.7229AnnexA coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.extra.audio.frames_per_pkt = 3;
t_Capability.capability = GCCAP_AUDIO_g729AnnexA;
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert 3rd capability parameter data for G.711Alaw 64kbit coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.capability = GCCAP_AUDIO_g711Alaw64k;
t_Capability.extra.audio.frames_per_pkt = 10;
```

```
/* For G.711 use frame size (ms) here, frames per packet fixed at 1 fpp */
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert fourth capability parameter data for G.711 Ulaw 64kbit coder */
/* changing only the coder type from previous capability */
t_Capability.capability = GCCAP_AUDIO_g711Ulaw64k;
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                            &t_Capability);

/* insert display string */
rc = gc_util_insert_parm_ref(&target_datap,
                             IPSET_CALLINFO,
                             IPPARM_DISPLAY,
                             (unsigned char)(strlen(IpDisplay)+1),
                             IpDisplay);

/* insert phone list */
rc = gc_util_insert_parm_ref(&target_datap,
                            IPSET_CALLINFO,
                            IPPARM_PHONELIST,
                            (unsigned char)(strlen(IpPhoneList)+1),
                            IpPhoneList);

/* insert user to user information */
rc = gc_util_insert_parm_ref(&target_datap,
                             IPSET_CALLINFO,
                             IPPARM_USERUSER_INFO,
                             (unsigned char)(strlen(IpUUI)+1),
                             IpUUI);

/* setting NS Data elements */
gc_util_insert_parm_ref(&target_datap,
                       IPSET_NONSTANDARDDATA,
                       IPPARM_NONSTANDARDDATA_DATA,
                       (unsigned char)(strlen(IpNSDataData)+1),
                       IpNSDataData);

if(ChoiceOfNSData) /* App chooses in advance which type of */
{                  /* second NS element to use */
   gc_util_insert_parm_ref(&target_datap,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_H221NONSTANDARD,
                           sizeof(IP_H221NONSTANDARD),
                           &appH221NonStd);
}

else
{
   gc_util_insert_parm_ref(&target_datap,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_NONSTANDARDDATA_OBJID,
                           (unsigned char)(strlen(IpCommonObjId)+1),
                           IpCommonObjId);
}
```

```
   /* setting NS Control elements */
   gc_util_insert_parm_ref(&target_datap,
                           IPSET_NONSTANDARDCONTROL,
                           IPPARM_NONSTANDARDDATA_DATA,
                           (unsigned char)(strlen(IpNSControlData)+1),
                           IpNSControlData);

   if(ChoiceOfNSControl) /* App chooses in advance which type of */
   {                     /* second NS element to use */
      gc_util_insert_parm_ref(&target_datap,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_H221NONSTANDARD,
                              sizeof(IP_H221NONSTANDARD),
                              &appH221NonStd);
   }
   else
   {
      gc_util_insert_parm_ref(&target_datap,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_NONSTANDARDDATA_OBJID,
                              (unsigned char)(strlen(IpCommonObjId)+1),
                              IpCommonObjId);
   }

   if(rc == 0)
   {
      gclib_mkbl.ext_datap = target_datap;
      rc = gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl,
                       MakeCallTimeout, EV_ASYNC);

      /* deallocate GC_PARM_BLK pointer */
      gc_util_delete_parm_blk(target_datap);
   }
}
```

SIP-Specific Code Example

The following code example shows how to make a call using the SIP protocol.

```
/* Make a SIP IP call on line device ldev */
void MakeSipIpCall(LINEDEV ldev)
{
   char *IpDisplay = "This is a Display";  /* display data */
   char *pDestAddrBlk = "12345@127.0.0.1"; /* destination IP address for MAKECALL_BLK */
   char *pSrcAddrBlk = "987654321"; /* origination address for MAKECALL_BLK*/

   int rc = 0;
   CRN crn;
   GC_MAKECALL_BLK gcmkbl;
   int MakeCallTimeout = 120;

   /* initialize GCLIB_MAKECALL_BLK structure */
   GCLIB_MAKECALL_BLK gclib_mkbl = {0};

   /* set to NULL to retrieve new parameter block from utility function */
   GC_PARM_BLK *target_datap = NULL;
   gcmkbl.cclib = NULL; /* CCLIB pointer unused */
   gcmkbl.gclib = &gclib_mkbl;

   /* set GCLIB_ADDRESS_BLK with destination string & type*/
   strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
   gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;
```

```
/* set GCLIB_ADDRESS_BLK with origination string & type*/
strcpy(gcmkbl.gclib->origination.address,pSrcAddrBlk);
gcmkbl.gclib->origination.address_type = GCADDRTYPE_TRANSPARENT;

/* set signaling PROTOCOL to SIP*/
rc = gc_util_insert_parm_val(&target_datap,
                             IPSET_PROTOCOL,
                             IPPARM_PROTOCOL_BITMASK,
                             sizeof(char),
                             IP_PROTOCOL_SIP);

/* initialize IP_CAPABILITY structure */
IP_CAPABILITY t_Capability = {0};
/* configure a GC_PARM_BLK with four coders, display, phone list and UUI message: */
/* specify and insert first capability parameter data for G.7231 coder */
t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLTRANSMIT;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLRECEIVE;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert second capability parameter data for G.7229AnnexA coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.extra.audio.frames_per_pkt = 3;
t_Capability.capability = GCCAP_AUDIO_g729AnnexA;
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert 3rd capability parameter data for G.711Alaw 64kbit coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.capability = GCCAP_AUDIO_g711Alaw64k;
t_Capability.extra.audio.frames_per_pkt = 10;

/* For G.711 use frame size (ms) here, frames per packet fixed at 1 fpp */
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);
```

```
   /* specify and insert fourth capability parameter data for G.711 Ulaw 64kbit coder */
   /* changing only the coder type from previous capability */
   t_Capability.capability = GCCAP_AUDIO_g711Ulaw64k;
   rc = gc_util_insert_parm_ref(&target_datap,
                                GCSET_CHAN_CAPABILITY,
                                IPPARM_LOCAL_CAPABILITY,
                                sizeof(IP_CAPABILITY),
                                &t_Capability);

   /* insert display string */
   rc = gc_util_insert_parm_ref(&target_datap,
                                IPSET_CALLINFO,
                                IPPARM_DISPLAY,
                                (unsigned char)(strlen(IpDisplay)+1),
                                IpDisplay);

   if (rc == 0)
   {
      gclib_mkbl.ext_datap = target_datap;
      /* numberstr parameter may be NULL if MAKECALL_BLK is set, as secondary
         address is ignored in SIP */
      rc = gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout,EV_ASYNC);

      /* deallocate GC_PARM_BLK pointer */
      gc_util_delete_parm_blk(target_datap);
   }
}
```

7.3.17 gc_OpenEx( ) Variances for IP

The **gc_OpenEx( )** function is supported in both synchronous and asynchronous mode, but the use of asynchronous mode is recommended.

The procedure for opening devices is the same regardless of whether H.323 or SIP is used. The IPT network device (N_ipt_BxTy) and IP Media device (M_ipmBxCy) can be opened in the same **gc_OpenEx( )** call and a voice device (V_dxxxBwCz) can also be included.

The format of the **devicename** parameter is:

```
:P_nnnn:N_iptBxTy:M_ipmBxCy:V_dxxxBwCz
```

Notes: **1.** The board and timeslot numbers for network devices do **not** have to be the same as the board and channel numbers for media devices.

2. It is possible to specify :N_iptBx (without any :M component) in the **devicename** parameter to get an IPT board device handle. Certain Global Call functions, such as **gc_SetConfigData( )**, use the IPT board device to specify call parameters (such as coders) for all devices in one operation or **gc_ReqService( )** to perform registration and deregistration operations. See Section 7.3.24, "gc_SetConfigData( ) Variances for IP", on page 334 and Section 7.3.21, "gc_ReqService( ) Variances for IP", on page 329 for more information.

3. It is also possible to specify :M_ipmBx (without any :N component) in the **devicename** parameter to get an IP Media board device handle.

The prefixes (P_, N_, M_ and V_) are used for parsing purposes. These fields may appear in any order. The conventions described below allow the Global Call API to map subsequent calls made on specific line devices or CRNs to interface-specific libraries. The fields within the **devicename** parameter must each begin with a colon.

The meaning of each field in the **devicename** parameter is as follows:

P_nnnn

Specifies the IP protocol to be used by the device. This field is mandatory. Possible values are:

- P_H323 – Use the device for H.323 calls only
- P_SIP – Use the device for SIP calls only
- P_IP – Multi-protocol option; use the device for SIP or H.323 calls

Note: When specifying an IPT board device (see below), use the multi-protocol option, P_IP.

N_iptBxTy

Specifies the name of the IPT network device where **x** is the logical board number and **y** is the logical channel number. An IPT board device can be specified using N_iptBx, where **x** is the logical board number.

M_ipmBxCy

Specifies the name of the IP Media device, where **x** is the logical board number and **y** is the logical channel number to be associated with an IPT network device. This field is optional.

V_dxxxBwCz

Specifies a voice resource, where **w** and **z** are the voice board and channel numbers respectively. This field is optional.

An IPT network device (iptBx) can also used for host LAN disconnect alarms. Note that all other Global Call alarms for IP are reported on IP Media (ipm) devices, not IPT network (ipt) devices.

Note: Applications should avoid closing and re-opening channels multiple times. Channels should be opened during initialization and should remain open for the duration of the application.

For Windows operating systems, the SRL function **sr_getboardcnt( )** can be used to retrieve the number of IPT board devices in the system. The **class_namep** parameter in this context should be DEV_CLASS_IPT. The SRL function **ATDV_SUBDEVS( )** can be used to retrieve the number of channels on a board. The **dev** parameter in this context should be an IPT board device handle, that is, a handle returned by **gc_OpenEx( )** when opening an IPT board device.

For Linux operating systems, the SRL device mapper functions **SRLGetAllPhysicalBoards( )**, **SRLGetVirtualBoardsOnPhysicalBoard( )** and **SRLGetSubDevicesOnVirtualBoard( )** can be used to retrieve information about the boards and devices in the system.

7.3.18 gc_RejectInitXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

Variance for H.323

The parameter **parmblkp** is ignored for IP technology and should be set to NULL.

The **gc_RejectInitXfer( )** function can be used at party C only on the receipt of GCEV_REQ_INIT_XFER.

Four of the six Global Call reasons are supported and result in the following CtIdentify error values signaled back to party A. Values GCVAL_REJREASON_INVADDR and GCVAL_REJREASON_INSUFFINFO cause the function to fail with a subsequent error code of IPERR_BAD_PARAM.

Table 27 lists the CtIdentity error codes that are signaled to party A based on the value of the **reason** parameter passed when the **gc_RejectXfer**( ) function is called.

Table 27. CtIdentify Errors Signaled From gc_RejectInitXfer( ) to the Network

GC Value	CtIdentify Error
GCVAL_REJREASON_INSUFFINFO	N/A (will return invalid parameter error)
GCVAL_REJREASON_INVADDR	N/A (will return invalid parameter error)
GCVAL_REJREASON_NOTALLOWED	suppServInteractionNotAllowed
GCVAL_REJREASON_NOTSUBSCRIBED	suppServInteractionNotAllowed
GCVAL_REJREASON_UNAVAIL	notAvailable
GCVAL_REJREASON_UNSPECIFIED	unspecified

Variance for SIP

This function does not apply to SIP call transfer. The SIP stack does not contact the Transfer Target or Transferred-To party (party C) until party A calls **gc_InvokeXfer**( ), so there is no issue of accepting or rejecting the transfer at the initiation stage.

7.3.19 gc_RejectXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

The parameter **parmblkp** is ignored for IP technology.

gc_RejectXfer( ) function can be used at party B only after the receipt of a GCEV_REQ_XFER event

Variance for H.323 (H.450.2)

All six Global Call rejection reasons are supported. Table 28 lists the CtInitiate error codes that are signaled to party A based on the value of the **reason** parameter passed when the **gc_RejectXfer**( ) function is called.

Table 28. CtInitiate Errors Signaled From gc_RejectXfer( ) to the Network

GC Value	CtInitiate Error
GCVAL_REJREASON_INSUFFINFO	invalidReroutingNumber
GCVAL_REJREASON_INVADDR	invalidReroutingNumber
GCVAL_REJREASON_NOTALLOWED	suppServInteractionNotAllowed

Table 28. CtInitiate Errors Signaled From gc_RejectXfer( ) to the Network (Continued)

GC Value	CtInitiate Error
GCVAL_REJREASON_NOTSUBSCRIBED	suppServInteractionNotAllowed
GCVAL_REJREASON_UNAVAIL	notAvailable
GCVAL_REJREASON_UNSPECIFIED	unspecified

Variance for SIP

The value of the **reason** parameter must be between IPEC_SIPReasonStatusMin and IPEC_SIPReasonStatusMax, as defined in the *gcip_defs.h* header file.

7.3.20 gc_ReleaseCallEx( ) Variances for IP

The **gc_ReleaseCallEx**( ) function is supported in both synchronous and asynchronous modes, but the use of asynchronous mode is recommended.

Note: An existing call on a line device must be released before an incoming call can be processed.

7.3.21 gc_ReqService( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_ReqService**( ) function can be used to register an endpoint with a registration server (gateway in H.323 or registrar in SIP). Function parameters must be set as follows:

target_type
GCTGT_GCLIB_NETIF

target_ID
An IPT board device, obtained by using **gc_OpenEx**( ) with a **devicename** parameter of "N_iptBx"

service_ID
Any valid reference to an unsigned long; must not be NULL

reqdatap
A pointer to a GC_PARM_BLK containing registration information.

respdatapp
Set to NULL for asynchronous mode. This function is not supported in synchronous mode.

mode
EV_ASYNC

The registration information that can be included is protocol specific as described in Table 29 and Table 30, below.

Registration options include:

- Overriding an existing registration value.
  In this case, IPPARM_OPERATION_REGISTER = IP_REG_SET_INFO.
- Adding a registration value.
  In this case, IPPARM_OPERATION_REGISTER = IP_REG_ADD_INFO.
- Removing a registration value; local alias or supported prefix only.
  In this case, IPPARM_OPERATION_REGISTER = IP_REG_DELETE_BY_VALUE.
- Querying a SIP Registrar for existing bindings (SIP only).
  In this case, IPPARM_OPERATION_REGISTER = IP_REG_QUERY_INFO.

See Section 4.23.4, "Code Examples", on page 229 for more information.

The **gc_ReqService( )** function also provides the following deregister options:

- Deregister and keep the registration information locally. In this case,
  IPPARM_OPERATION_DEREGISTER = IP_REG_MAINTAIN_LOCAL_INFO
- Deregister and discard the registration information locally.
  In this case, IPPARM_OPERATION_DEREGISTER = IP_REG_DELETE_ALL

See Section 4.23.4.2, "Deregistration Example", on page 231 for more information.

Since some of the registration data may be protocol specific, there is a facility to set the protocol type using IP parameters in **reqdatap** and **respdatapp**, which are of type GC_PARM_BLK.

The relevant items for the GC_PARM_BLK are the IPSET_PROTOCOL parameter set ID and the IPPARM_PROTOCOL_BITMASK parameter ID with one of the following values:

- IP_PROTOCOL_H323
- IP_PROTOCOL_SIP
- IP_PROTOCOL_H323 | IP_PROTOCOL_SIP

Note: The default value for the protocol, when not specified by the application, is IP_PROTOCOL_H323.

The GCEV_SERVICERESP event, which is received on an IPT board device handle, indicates that a service request has been responded to by an H.323 gatekeeper or a SIP registrar. This event does not necessarily mean that the registration operation itself was completed successfully, however; successfulr completion of the operation is indicated by the result code IPERR_OK. The event data includes a specification of the protocol used, using the IPSET_PROTOCOL parameter set ID and the IPPARM_PROTOCOL_BITMASK parameter ID with one of the following values:

- IP_PROTOCOL_H323
- IP_PROTOCOL_SIP

Variance for H.323

When using H.323, the registration information that can be included in the GC_PARM_BLK associated with the **gc_ReqService( )** function is shown in Table 29.

Table 29. Registration Information When Using H.323

Set ID	Parameter IDs and Values
GCSET_SERVREQ	PARM_REQTYPE † Data type: IP_REQTYPE_REGISTRATION
GCSET_SERVREQ	PARM_ACK †
IPSET_PROTOCOL	IPPARM_PROTOCOL_BITMASK Bitmask composed from one or both of the following values: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP
IPSET_REG_INFO See Section 8.2.20, “IPSET_REG_INFO”, on page 369, for more information.	IPPARM_OPERATION_REGISTER, with defined values: • IP_REG_SET_INFO • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE IPPARM_OPERATION_DEREGISTER, with defined values: • IP_REG_MAINTAIN_LOCAL_INFO • IP_REG_DELETE_ALL IPPARM_REG_ADDRESS Data type IP_REGISTER_ADDRESS See the reference page for IP_REGISTER_ADDRESS on page 392 for more information IPPARM_REG_TYPE, with defined values: • IP_REG_GATEWAY • IP_REG_TERMINAL
IPSET_LOCAL_ALIAS	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_H323_ID IPPARM_ADDRESS_PHONE IPPARM_ADDRESS_TRANSPARENT IPPARM_ADDRESS_URL Data type: String
IPSET_SUPPORTED_PREFIXES	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_H323_ID IPPARM_ADDRESS_PHONE IPPARM_ADDRESS_TRANSPARENT IPPARM_ADDRESS_URL Data type: String
† indicates mandatory parameters. These parameters are required to support the generic service request mechanism provided by Global Call and are not sent in any registration message.	

Multiple aliases and supported prefix information is supported when the target protocol for registration is H.323.

Variance for SIP

When using SIP, the registration information that can be included in the GC_PARM_BLK associated with the **gc_ReqService( )** function is shown in Table 30.

Table 30. Registration Information When Using SIP

Set ID	Parameter IDs
GCSET_SERVREQ	PARM_REQTYPE † Data type IP_REQTYPE_REGISTRATION
GCSET_SERVREQ	PARM_ACK †
IPSET_LOCAL_ALIAS	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_TRANSPARENT Data type: String
IPSET_PROTOCOL	IPPARM_PROTOCOL_BITMASK Bitmask composed from one or both of the following values: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP
IPSET_REG_INFO See Section 8.2.20, “IPSET_REG_INFO”, on page 369, for more information.	IPPARM_OPERATION_REGISTER, with defined values: • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE • IP_REG_QUERY_INFO • IP_REG_SET_INFO IPPARM_OPERATION_DEREGISTER, with defined values: • IP_REG_MAINTAIN_LOCAL_INFO • IP_REG_DELETE_ALL IPPARM_REG_ADDRESS Datatype IP_REGISTER_ADDRESS See the reference page for IP_REGISTER_ADDRESS on page 392 for more information IPPARM_REG_AUTOREFRESH, with defined values: • IP_REG_AUTOREFRESH_DISABLE • IP_REG_AUTOREFRESH_ENABLE
† indicates mandatory parameters. These parameters are required to support the generic service request mechanism provided by Global Call and are not sent in any registration message.	

Multiple aliases are supported when the target protocol for registration is SIP, but prefix information is **ignored**.

When using SIP, auto-refresh is enabled by default if there is no IPSET_REG_INFO / IPPARM_REG_AUTOREFRESH parameter specified. The default for the requested expiration time is 3600 seconds; the actual expiration time is determined by the Registrar.

7.3.22 gc_RespService( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_RespService( )** function operates on an IPT board device and is used to respond to requests from an H.323 gatekeeper or a SIP registrar. Since some of the data may be protocol specific (in future releases), there is a facility to set the protocol type using IP parameters in **datap**, which is of type GC_PARM_BLK.

The following are the relevant function parameters:

target_type
GCTGT_CCLIB_NETIF

target_id
IPT board device

The relevant items for the GC_PARM_BLK are the IPSET_PROTOCOL parameter set ID and the IPPARM_PROTOCOL_BITMASK parameter ID with one of the following values:

- IP_PROTOCOL_H323
- IP_PROTOCOL_SIP
- IP_PROTOCOL_H323 | IP_PROTOCOL_SIP

Note: The default value for the protocol, when not specified by the application, is IP_PROTOCOL_H323.

The GCEV_SERVICEREQ event indicates that a service has been requested by an H.323 gatekeeper or a SIP registrar. The event is received on an IPT board device handle. The event data includes a specification of the protocol used, using the IPSET_PROTOCOL parameter set ID and the IPPARM_PROTOCOL_BITMASK parameter ID with one of the following values:

- IP_PROTOCOL_H323
- IP_PROTOCOL_SIP

7.3.23 gc_SetAlarmParm( ) Variances for IP

The **gc_SetAlarmParm( )** function can be used to set QoS threshold values. The function parameter values in this context are:

linedev
The media device handle, retrieved using the **gc_GetResourceH( )** function. See Section 4.22.2, "Retrieving the Media Device Handle", on page 215 for more information.

aso_id
The alarm source object ID. Set to ALARM_SOURCE_ID_NETWORK_ID.

ParmSetID
Must be set to ParmSetID_qosthreshold_alarm.

alarm_parm_list
A pointer to an ALARM_PARM_FIELD structure. The alarm_parm_number field is not used. The alarm_parm_data field is of type GC_PARM, which is a union. In this context, the type used is void *pstruct, and is cast as a pointer to an IPM_QOS_THRESHOLD_INFO structure,

which includes an IPM_QOS_THRESHOLD_DATA structure that contains the parameters representing threshold values. See the IPM_QOS_THRESHOLD_INFO data structure pages in the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information.

The thresholds supported by Global Call for HMP are:

- QOSTYPE_JITTER
- QOSTYPE_LOSTPACKETS
- QOSTYPE_RTCPTIMEOUT
- QOSTYPE_RTPTIMEOUT

mode

Must be set to EV_SYNC.

Note: Applications **must** include the *gcipmlib.h* header file before Global Call can be used to set or retrieve QoS threshold values.

See Section 4.22.3, “Setting QoS Threshold Values”, on page 215 for code examples.

7.3.24 gc_SetConfigData( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_SetConfigData( )** function is used for a number of different purposes:

- setting parameters for all board devices, including devices that are already open
- enabling and disabling unsolicited GCEV_EXTENSION events on a board device basis
- setting the type of DTMF support and the RFC 2833 payload type on a board device basis
- setting T.38 fax server operating mode
- masking and unmasking call state events on a line device basis

Notes:

1. The **gc_SetConfigData( )** function operates on board devices, that is, devices opened using **gc_OpenEx( )** with :N_iptBx:P_IP in the **devicename** parameter. By its nature, a board device is multi-protocol, that is, it applies to both the H.323 and SIP protocols and is not directed to one specific protocol. You *cannot* open a board device (with :P_H323 or :P_SIP in the **devicename** parameter) to target a specific protocol.
2. When using the **gc_SetConfigData( )** function to set parameters, the parameter values apply to all board devices, including devices that are already open. The parameters can be overridden by specifying new values in the **gc_SetUserInfo( )** function (on a per line device basis) or the **gc_MakeCall( )** function (on a per call basis).
3. Coder information can be specified for a device when using **gc_SetConfigData( )**, or when using **gc_MakeCall( )** to make a call, or when using **gc_AnswerCall( )** to answer a call.
4. Use **gc_SetUserInfo( )** to set parameters on line devices.

When using the **gc_SetConfigData( )** function on a board device (the first four bullets above), use the following function parameter values:

target_type

GCTGT_CCLIB_NETIF

target_id

An IPT board device that can be obtained by using the **gc_OpenEx( )** function with :N_iptBx:P_IP in the **devicename** parameter. See Section 7.3.17, “gc_OpenEx( ) Variances for IP”, on page 326 for more information.

target_datap

A pointer to a GC_PARM_BLKP structure that contains the parameters to be configured. The parameters that can be included in the GC_PARM_BLK are protocol specific. See the following “Variance for H.323” and “Variance for SIP” sections.

As in other technologies supported by Global Call, the **gc_SetConfigData( )** function can be used to mask call state events, such as GCEV_ALERTING, on a line device basis. When used for this purpose, the **target_type** is GCTGT_GCLIB_CHAN and the **target_ID** is a line device. See the “Call State Event Configuration” section in the *Global Call API Programming Guide* for more information on masking events in general.

Variance for H.323

Table 29 describes the call parameters that can be included in the GC_PARM_BLK associated with the **gc_SetConfigData( )** function. These parameters are in addition to the call parameters described in Table 25, “Configurable Call Parameters When Using H.323”, on page 312 that can also be included.

Table 31. Parameters Configurable Using gc_SetConfigData( ) When Using H.323

Set ID	Parameter IDs	Use Before †
GCSET_CALL_CONFIG	GCPARM_CALLPROC †† Enumeration with one of the following values: • GCCONTROL_APP – The application must use **gc_CallAck( )** to send the Proceeding message. This is the default. • GCCONTROL_TCCL – The stack sends the Proceeding message automatically.	**gc_AnswerCall( )**
IPSET_CALLINFO	IPPARM_H245TUNNELING ††† Enumeration with one of the following values: • IP_H245TUNNELINGON • IP_H245TUNNELINGOFF	**gc_AnswerCall( )**

IPSET_CONFIG	IPPARM_OPERATING_MODE
	Enumeration with one of the following values:
	• IP_AUTOMATIC_MODE
	• IP_MANUAL_MODE
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_DTMF	IPPARM_SUPPORT_DTMF_BITMASK
	Datatype: Uint8[ ]
	IPPARM_DTMF_RFC2833_PAYLOAD_TYPE
	Datatype: Uint8[ ]
		gc_AnswerCall( )
		gc_MakeCall( )

† Information can be set in any state but it is only used in certain states. See the “variances” section for the specific function for more information.
†† This is a system configuration parameter for the terminating side, not a call configuration parameter. It cannot be overwritten by setting a new value in **gc_SetUserInfo( )** or **gc_MakeCall( )**.
††† Applies to the configuration of tunneling for inbound calls only. See Section 4.19, “Enabling and Disabling H.245 Tunneling”, on page 206 for more information.

Table 31. Parameters Configurable Using gc_SetConfigData( ) When Using H.323 (Continued)

Set ID	Parameter IDs	Use Before †

IPSET_VENDORINFO	IPPARM_VENDOR_PRODUCT_ID String, max. length = MAX_PRODUCT_ID_LENGTH (32)
	IPPARM_VENDOR_VERSION_ID String, max. length = MAX_VERSION_ID_LENGTH (32)
	IPPARM_H221NONSTD Datatype IP_H221NONSTANDARD.
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_EXTENSIONEVT_MSK	GCACT_ADDMSK Datatype: Uint8[ ] GCACT_SETMSK Datatype: Uint8[ ] GCACT_SUBMSK Datatype: Uint8[ ]	**gc_AnswerCall( )**

† Information can be set in any state but it is only used in certain states. See the "variances" section for the specific function for more information.
†† This is a system configuration parameter for the terminating side, not a call configuration parameter. It cannot be overwritten by setting a new value in **gc_SetUserInfo( )** or **gc_MakeCall( )**.
††† Applies to the configuration of tunneling for inbound calls only. See Section 4.19, "Enabling and Disabling H.245 Tunneling", on page 206 for more information.

Variance for SIP

The **gc_SetConfigData( )** function can be used to enable and disable the optional GCEV_INVOKEXFER_ACCEPTED event on a line device basis. This event is only relevant when the call transfer supplementary service is enabled, and is generated to notify the Transferor or Transferring application (party A) that the Transferee or Transferred party (party B) has received and accepted a call transfer request. As with other maskable call state events, the parameter set ID to use is GCSET_CALLEVENT_MSK, and the parameter IDs that may be used are GCACT_ADDMSK, GCACT_SUBMSK, and GCACT_SETMSK. The specific parameter value that is used to enable or disable the GCEV_INVOKEXFER_ACCEPTED event is GCMSK_INVOKEXFER_ACCEPTED. Note that there is no corresponding event for H.450.2 call transfers.

Table 32 describes the call parameters that can be included in the GC_PARM_BLK associated with the **gc_SetConfigData( )** function. These parameters are in addition to the call parameters described in Table 26, "Configurable Call Parameters When Using SIP", on page 314 that can also be included.

Table 32. Parameters Configurable Using gc_SetConfigData( ) When Using SIP

Set ID	Parameter IDs	Use Before †
GCSET_CALL_CONFIG	GCPARM_CALLPROC †† Enumeration with one of the following values: • GCCONTROL_APP – The application must use **gc_CallAck( )** to send the Proceeding message. This is the default. • GCCONTROL_TCCL – The stack sends the Proceeding message automatically.	**gc_AnswerCall( )**

IPSET_CONFIG	IPPARM_OPERATING_MODE
	Enumeration with one of the following values:
	• IP_AUTOMATIC_MODE
	• IP_MANUAL_MODE
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_DTMF	IPPARM_SUPPORT_DTMF_BITMASK
	Datatype: Uint8[ ]
	IPPARM_DTMF_RFC2833_PAYLOAD_TYPE
	Datatype: Uint8[ ]
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_EXTENSIONEVT_MSK	GCACT_ADDMSK Datatype: Uint8[ ] GCACT_SETMSK Datatype: Uint8[ ] GCACT_SUBMSK Datatype: Uint8[ ]	**gc_AnswerCall( )**

† Information can be set in any state but it is only used in certain states. See the “variances” section for the specific function for more information.
†† This is a system configuration parameter for the terminating side, not a call configuration parameter. It cannot be overwritten by setting a new value in **gc_SetUserInfo( )** or **gc_MakeCall( )**.

7.3.25 gc_SetUserInfo( ) Variances for IP

The **gc_SetUserInfo**( ) function can be used to:

- set call values for all calls on the specified line device
- set call values for the duration of a single call
- set SIP message information fields
- associate and disassociate a T.38 Fax device with a Media device

The **gc_SetUserInfo**( ) function is used only to set the values of call-related information, such as coder information, display information, phone list, etc. before a call has been initiated. The information is not transmitted until the next Global Call function that initiates the transmission of information on the line, such as, **gc_AnswerCall**( ), **gc_AcceptCall**( ), or **gc_CallAck**( ).

Note: If no coder type is specified, all supported coder types will be used in the coder negotiation.

The parameters that are configurable using **gc_SetUserInfo**( ) are given in Table 25, “Configurable Call Parameters When Using H.323”, on page 312 and Table 26, “Configurable Call Parameters

When Using SIP", on page 314. In addition, the DTMF support bitmask, (see Table 31 and Table 32) is also configurable using **gc_SetUserInfo( )**.

Note: The **gc_SetUserInfo( )** function may **not** be used to set the IP protocol for a multi-protocol line device (i.e., one that was opened in P_IP mode). The only mechanism for selecting the protocol to use is the GC_MAKECALL_BLK structure associated with the **gc_MakeCall( )** function.

The **gc_SetUserInfo( )** function operates on either a CRN or a line device:

- If the target of the function is a CRN, the information in the function is automatically directed to the protocol associated with that CRN.
- If the target of the function is a line device, then:
  - If the line device was opened as a multi-protocol device (:N_PIP), the information in the function is automatically directed to each protocol and is used by either H.323 or SIP calls made subsequently.
  - If the line device was opened as a single-protocol device (:N_H323 or :N_SIP), then the information in the function automatically applies to that protocol only and is used by calls made using that protocol.

Note: Use **gc_SetConfigData( )** to set parameters on board devices.

The **gc_SetUserInfo( )** is also used to set Information Elements (IEs) in Q.931 messages. See Section 4.5.3, "Setting Q.931 Message IEs", on page 128 for more information.

7.3.25.1 Setting Call Parameters for the Next Call

The relevant function parameter values in this context are:

target_type
GCTGT_GCLIB_CRN (if a CRN exists) or GCTGT_GCLIB_CHAN (if a CRN does not exist)

target_id
CRN (if it exists) or line device (if a CRN does not exist)

duration
GC_SINGLECALL

infoparmblkp
a pointer to a GC_PARM_BLK with a list of parameters (including coder information) to be set for the line device.

Note: If a call is in the Null state, the new parameter values apply to the next call. If a call is in a non-Null state, the new parameter values apply to the remainder of the current call only.

7.3.25.2 Setting Call Parameters for the Next and Subsequent Calls

When the **duration** parameter is set to GC_ALLCALLS, the new call values become the default values for the line device and are used for all subsequent calls on that device. The pertinent function parameter values in this context are:

target_type
GCTGT_GCLIB_CHAN

target_id
line device

duration
GC_ALLCALLS

infoparmblkp
a pointer to a GC_PARM_BLK with a list of parameters (including coder information) to be set for the line device.

Note: If a call is in the Null state, the new parameter values apply to the next call and all subsequent calls. If a call is in a non-Null state, the new parameter values apply to the remainder of the current call and all subsequent calls.

7.3.25.3 Setting SIP Message Information Fields

The **gc_SetUserInfo( )** function can be used to set SIP message information fields. The relevant function parameter values in this context are:

target_type
GCTGT_GCLIB_CHAN

target_id
line device

duration
GC_SINGLECALL

infoparmblkp
A pointer to a GC_PARM_BLK that contains the IPSET_SIP_MSGINFO parameter set ID and one of the following parameter IDs that identify the fields to be set:

- IPPARM_CALLID_HDR
- IPPARM_CONTACT_DISPLAY
- IPPARM_CONTACT_URI
- IPPARM_DIVERSION_URI
- IPPARM_FROM_DISPLAY
- IPPARM_REFERRED_BY
- IPPARM_REPLACES
- IPPARM_REQUEST_URI
- IPPARM_TO_DISPLAY

See Section 4.6.5, “Setting SIP Header Fields for Outbound Messages”, on page 141 for more information and a cod e example.

7.3.25.4 Associating and Disassociating a T.38 Fax Device with a Media Device

To support T.38 fax server operation, the **gc_SetUserInfo( )** function is used to associate a T.38 Fax device with a Media device to facilitate a switch from an audio session to a T.38 fax session. Similarly, when switching form a T.38 fax session to an audio session, the **gc_SetUserInfo( )** function is used to disassociate the T.38 Fax device from the Media device. The relevant function parameter values in this context are:

target_type
GCTGT_GCLIB_CRN

target_id
CRN

duration
GC_SINGLECALL

infoparmblkp
a pointer to a GC_PARM_BLK that contains:

- the IPSET_FOIP parameter set ID and one of the following parameter IDs:
  – IPPARM_T38_CONNECT when switching from audio to T.38 fax
  – IPPARM_T38_DISCONNECT when switching from T.38 fax to audio
- an associated IP_CONNECT structure that contains the fax and media handles and the connection type (half-duplex or full-duplex)

See Section 4.26.3, “Initiating a Switch from Audio to T.38 Fax”, on page 249 for more information and a code example.

7.3.26 gc_Start( ) Variances for IP

The **gc_Start( )** function is used to define the number of IPT board devices to create (see Section 2.3.2, “IPT Board Devices”, on page 43 for the meaning of an IPT board device) and the parameters for each IPT board device. Among the major characteristics that can be configured for each virtual board when starting the system are:

- the total number of IPT devices that can be open concurrently
- the maximum number of IPT devices that can be used for H.323 calls and for SIP calls
- the local address and signaling port for H.323 and for SIP
- enable/disable call transfer supplementary services
- enable/disable access to H.323 message information fields and to SIP message header fields
- enable/disable and configure access to MIME-encoded message bodies in SIP messages
- enable/disable and configure SIP outbound proxy
- enable/disable and configure use of TCP transport protocol for SIP messages
- configure SIP request retry behavior
- enable/disable application access to SIP OPTIONS messages

If NULL is passed to **gc_Start( )** the system is started in a default configuration that has two virtual boards, one for H.323 and one for SIP. Each of these board devices has the default parameters listed at the end of this section. If the default two-board configuration is not appropriate for the application, or if the application requires a non-default configuration for any of the parameters (for example, if it needs to use one or more of the features that are disabled by default), the application must explicitly configure the system before calling **gc_Start( )**.

To configure a non-default system, the application starts by creating an IPCCLIB_START_DATA structure and an array of IP_VIRTBOARD structures, one for each virtual board in the system. The application **must** then use the convenience functions **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** (defined in the *gcip.h* header file) to initialize each of the structures with the default value for each field in the structure. After initialization, the application can override the default value for any fields in any of these data structures to configure the virtual boards as desired. After the fields in the IPCCLIB_START_DATA and IP_VIRTBOARD structures have been configured, the IPCCLIB_START_DATA structure is passed to **gc_Start( )** via pointers in CCLIB_START_STRUCT and GC_START_STRUCT data structures.

As a very simple example, the following code illustrates the **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** convenience functions being used to initialize the data structures for a two-board system and the default IP_VIRTBOARD structures being modified to enable the call transfer supplementary service:

```
IP_VIRTBOARD ip_virtboard[2];
IPCCLIB_START_DATA ipcclibstart;
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
```

When calling **gc_Start( )** with configuration data that has been set by the application, the array of CCLIB_START_STRUCT structures that is pointed to by GC_START_STRUCT must include two mandatory members to start the libraries for IP call control signaling and for IP media devices. One of these structures contains “GC_IPM_LIB” as the cclib_name field and NULL as the cclib_data field. The other structure contains “GC_H3R_LIB” as cclib_name and a pointer to the configured IPCCLIB_START_DATA structure as cclib_data..

The total number of IPT devices is not necessarily the number of IPT devices used for H.323 calls plus the number of IPT devices used for SIP calls. Each IPT device can be used for both H.323 and SIP. If there are 2016 devices available (total_max_calls=2016, three Intel NetStructure IPT boards), you can specify that all 2016 devices can be used for both H.323 calls (max_h323=2016) and SIP (max_sip=2016), or half are used for H.323 only (max_h323=1008) and half are used for SIP only (max_sip=1008), or any other such combination.

The default value for the maximum number of IPT devices is 120, but this can be set to a value up to 2016. See the reference page for IP_VIRTBOARD on page 394 for more information. The local IP address for each IPT board device is a parameter of type IPADDR in the IP_VIRTBOARD structure. See the reference page for IP_ADDR on page 382 for more information.

Notes: ***1.*** When using Global Gall over IP, the GC_LIB_START structure must include both the GC_H3R_LIB and GC_IPM_LIB libraries since there are inter-dependencies.

2. The maximum value of the num_boards field is 8.

The total_max_calls, h323_max_calls, and SIP_max_calls fields in the IP_VIRTBOARD structure can be used to allocate the number and types of calls among the available devices. The following #defines have been provided as a convenience to application developers:

IP_CFG_DEFAULT
: indicates to the call control library that it should determine and fill in the correct value

IP_CFG_MAX_AVAILABLE_CALLS
: indicates to the call control library that it should use the maximum available resources

> ***Note:*** Do not use the value IP_CFG_MAX_AVAILABLE_CALLS with applications running on HMP 1.1. That value initializes the stack for 2016 channels, which results in a lengthy initialization time and is an inefficient use of memory and other system resources.

IP_CFG_NO_CALLS
: indicates to the call control library that it should not allocate **any** resources

The following restrictions apply when overriding values in the IPCCLIB_START_DATA structure. The **gc_Start( )** function will fail if these restrictions are not observed.

- The total number of devices (total_max_calls) must not be larger than the sum of the values for the maximum number of H.323 calls (h323_max_calls) and the maximum number of SIP calls (sip_max_calls).
- The total number of devices (total_max_calls) cannot be set to IP_CFG_NO_CALLS.
- The maximum number of H.323 calls (h323_max_calls) and maximum number of SIP calls (sip_max_calls) values cannot both be set to IP_CFG_NO_CALLS.
- When configuring multiple board devices, IP_CFG_DEFAULT cannot be used as an address specifier.
- If different IP addresses or port numbers are not used when running multiple instances of an application for any one technology (H.323 or SIP), then the xxx_max_calls (xxx = h323 or sip) parameter for the other technology must be set to IP_CFG_NO_CALLS.

Default configuration parameter values

The following parameter values are set for each of two virtual boards (one for H.323 and one for SPI) if NULL is passed to **gc_Start( )**. If this two-board configuration is not appropriate, or if the application requires any of the disabled features to be enabled, it must define and initialize an IPCCLIB_START_DATA structure and an array of IP_VIRTBOARD structures, then override the default values as necessary before passing the information to **gc_Start( )**.

The following parameters that are set in the IPCCLIB_START_DATA structure apply to both boards:

- delimiter = , [default parsing delimiter for address strings is a comma]
- max_parm_data_size = 256

The parameters for the default H.323 virtual board are:

- total_max_calls = 120
- h323_max_calls = 120

- h323_signaling_port = 1720
- localIP.ip_ver = IPVER4
- localIP.u_ipaddr.ipv4 — determined by socket functions
- h323_msginfo_mask = IP_H323_MSGINFO_DISABLE
- sup_serv_mask = IP_SUP_SERV_DISABLED
- terminal_type = IP_TT_GATEWAY

The parameters for the default SIP virtual board are:

- total_max_calls = 120
- sip_max_calls = 120
- sip_signaling_port = 5060
- localIP.ip_ver = IPVER4
- localIP.u_ipaddr.ipv4 — determined by socket functions
- sip_msg_info_mask = IP_SIP_MSGINFO_DISABLE
- sup_serv_mask = IP_SUP_SERV_DISABLED
- sip_mime_mem = Disabled
- outbound_proxy_IP = Disabled
- outbound_proxy_port = 5060
- outbound_proxy_hostname = NULL
- terminal_type = N/A (not used)
- E_SIP_tcpenabled = ENUM_Disabled
- E_SIP_OutboundProxyTransport = ENUM_UDP
- E_SIP_Persistence = ENUM_PERSISTENCE_TRANSACT_USER
- SIP_maxUDPmsgLen = 1300
- E_SIP_DefaultTransport = UNUM_UDP
- E_SIP_RequestRetry = ENUM_REQUEST_RETRY_ALL
- E_SIP_OPTIONS_Access = ENUM_Disabled

7.3.27 gc_UnListen( ) Variances for IP

The **gc_UnListen( )** function is supported in both synchronous and asynchronous modes. The function is blocking in synchronous mode.

Note: For line devices that comprise media (ipm) and voice (dxxx) devices, routing is only done on the media devices. Routing of the voice devices must be done using the Voice API (dx_ functions).

7.4 Global Call States Supported by IP

The following Global Call call states are supported when using Global Call with IP technology:

- GCST_ACCEPTED

- GCST_ACCEPT_XFER
- GCST_ALERTING
- GCST_CALLROUTING
- GCST_CONNECTED
- GCST_DETECTED
- GCST_DIALING
- GCST_DISCONNECTED
- GCST_IDLE
- GCST_INVOKE_XFER_ACCEPTED
- GCST_INVOKE_XFER
- GCST_NULL
- GCST_OFFERED
- GCST_PROCEEDING
- GCST_REQ_INIT_XFER
- GCST_REQ_XFER
- GCST_XFER_CMPLT

See the *Global Call API Programming Guide* for more information about the call state models.

7.5 Global Call Events Supported by IP

The following Global Call events are supported when using Global Call with IP technology:

- GCEV_ACCEPT
- GCEV_ACCEPT_INIT_XFER (H.323/H.450.2 only)
- GCEV_ACCEPT_INIT_XFER_FAIL (H.323/H.450.2 only)
- GCEV_ACCEPT_XFER
- GCEV_ACCEPT_XFER_FAIL
- GCEV_ACKCALL (deprecated; equivalent is GCEV_CALLPROC)
- GCEV_ALARM
- GCEV_ALERTING (maskable)
- GCEV_ANSWERED
- GCEV_ATTACH
- GCEV_ATTACHFAIL
- GCEV_BLOCKED
- GCEV_CONNECTED
- GCEV_CALLPROC
- GCEV_DETECTED (maskable)
- GCEV_DETACH

- GCEV_DETACHFAIL
- GCEV_DIALING (maskable)
- GCEV_DISCONNECTED
- GCEV_DROPCALL
- GCEV_ERROR
- GCEV_EXTENSION (unsolicited event)
- GCEV_EXTENSIONCMPLT (termination event for **gc_Extension( )**)
- GCEV_FATALERROR
- GCEV_INIT_XFER
- GCEV_INIT_XFER_FAIL (H.323/H.450.2 only)
- GCEV_INIT_XFER_REJ (H.323/H.450.2 only)
- GCEV_INVOKE_XFER
- GCEV_INVOKE_XFER_ACCEPTED (maskable, SIP only)
- GCEV_INVOKE_XFER_FAIL
- GCEV_INVOKE_XFER_REJ
- GCEV_LISTEN
- GCEV_OFFERED
- GCEV_OPENEX
- GCEV_OPENEX_FAIL
- GCEV_PROCEEDING (maskable)
- GCEV_REJ_INIT_XFER (H.323/H.450.2 only)
- GCEV_REJ_INIT_XFER_FAIL (H.323/H.450.2 only)
- GCEV_REJ_XFER
- GCEV_REJ_XFER_FAIL
- GCEV_RELEASECALL
- GCEV_REQ_INIT_XFER (H.323/H.450.2 only)
- GCEV_REQ_XFER
- GCEV_RESETLINEDEV
- GCEV_SERVICEREQ
- GCEV_SERVICERESP
- GCEV_SERVICERESPCMPLT
- GCEV_SETCONFIGDATA
- GCEV_SETCONFIGDATAFAIL
- GCEV_TASKFAIL
- GCEV_UNBLOCKED
- GCEV_UNLISTEN
- GCEV_XFER_CMPLT
- GCEV_XFER_FAIL

See the *Global Call API Library Reference* for more information about Global Call events.

IP-Specific Parameters 8

This chapter describes the parameter set IDs (set IDs) and parameter IDs (parm IDs) used with IP technology. Topics include:

8.1 Overview of Parameter Usage

The parameter set IDs and parameter IDs described in this chapter are defined in the *gcip.h* header file. Table 33 summarizes the parameter sets and parameters used by Global Call in an IP environment, organized alphabetically by set ID and then by parameter ID.

The meaning of the columns in Table 33 are:

- **Set ID** – An identifier for a group of related parameters.
- **Parameter ID** – An identifier for a specific parameter.
- **Set** – Indicates the Global Call functions used to set the parameter information.
- **Send** – Indicates the Global Call functions used to send the information to a peer endpoint.
- **Retrieve** – Indicates the Global Call function used to retrieve information that was sent by a peer endpoint.
- **H.323/SIP** – Indicates if the parameter is supported when using H.323, SIP, or both.

Detailed information about each of the parameters in each parameter set is provided in the second part of this chapter.

Table 33. Summary of Parameter Sets and Parameter Usage

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
GCSET_ CALL_CONFIG	GCPARM_ CALLPROC	**gc_SetConfigData( )**	---	---	both

GCSET_ CHAN_ CAPABILITY	IPPARM_ LOCAL_CAPABILITY	**gc_SetConfigData( )**
		gc_SetUserInfo( ) †
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	both

IPSET_ CALLINFO	IPPARM_ BEARERCAP	**gc_SetUserInfo( )** (GC_SINGLECALL only)	**gc_MakeCall( )**	Retrieve from GCEV_OFFERED event via **gc_GetMetaEvent( )**	H.323 only
	IPPARM_ CALLDURATION	---	---	**gc_Extension( )** (IPEXTID_GETINFO)	both

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323

IPSET_ CALLINFO	IPPARM_CALLID	**gc_MakeCall( )**
		gc_SetUserInfo( ) (GC_SINGLECALL only)
			gc_MakeCall( )	**gc_GetCallInfo( )** (IP_CALLID)
				–or– **gc_Extension( )** (IPEXTID_GETINFO)
				Note: The use of **gc_Extension( )** to retrieve the Call ID is being deprecated; use **gc_GetCallInfo( )**.
					both

	IPPARM_ CONNECTION METHOD	**gc_MakeCall( )**
		gc_SetUserInfo( ) †
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	both

	IPPARM_DISPLAY	**gc_SetUserInfo( )** †
		gc_MakeCall( )
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	both

	IPPARM_FACILITY	**gc_SetUserInfo( )** (GC_SINGLECALL only)	**gc_AnswerCall( )**
			gc_MakeCall( )
				In GCEV_OFFERED, GCEV_CONNECTED, or GCEV_EXTENSION event (with ext_id of IPEXTID_ RECEIVEMSG). Retrieve IE value with **gc_GetMetaEvent( )**	H.323 only

	IPPARM_ H245TUNNELING	**gc_SetUserInfo( )** † **gc_MakeCall( )** **gc_SetConfigData( )** ‡	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323 only

	IPPARM_MEDIA WAITFORCONNECT	**gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_GetMetaEvent( )** (GCEV_OFFERED)
				gc_util_next_parm( )
					H.323 only

	IPPARM_ PHONELIST	**gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	both

	IPPARM_ PRESENTATION_IND	**gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_GetMetaEvent( )** (GCEV_OFFERED)
				gc_util_next_parm( )
					H.323 only

	IPPARM_ PROGRESS_IND	---	---	**gc_GetMetaEvent( )** (GCEV_EXTENSION)
				gc_util_next_parm( )
				Note: Extension events associated with Progress messages are masked by default. Enable them via gc_SetUserInfo(IPSET_EXTENSIONEVT_MSK, GCACT_SETMSK, EXTENSIOEVT_ CALL_PROGRESS)
					H.323 only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ CALLINFO	IPPARM_ USERUSER_INFO	**gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323 only

IPSET_ CONFERENCE	IPPARM_ CONFERENCE_ GOAL	**gc_MakeCall( )**
		gc_SetUserInfo( ) †
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

	IPPARM_ CONFERENCE_ID	---	---	**gc_Extension( )** (IPEXTID_GETINFO)	H.323 only
IPSET_CONFIG	IPPARM_ AUTHENTICATION_ CONFIGURE	**gc_SetAuthentication Info( )**	---	---	SIP only
	IPPARM_ AUTHENTICATION_ REMOVE	**gc_SetAuthentication Info( )**	---	---	SIP only

	IPPARM_ CONFIG_TOS	**gc_MakeCall( )**
		gc_SetUserInfo( ) †
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	both

	IPPARM_ OPERATING_MODE	**gc_SetConfigData( )**	---	---	both
	IPPARM_ REGISTER_ SIP_HEADER	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	---	---	SIP only
IPSET_DTMF	IPPARM_ DTMF_ ALPHANUMERIC	---	**gc_Extension( )** (IPEXTID_SEND_ DTMF)	**gc_Extension( )** (IPEXTID_ RECEIVE_DTMF)	both
	IPPARM_ DTMF_RFC2833_ PAYLOAD_TYPE	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	---	---	both
	IPPARM_ SUPPORT_DTMF_ BITMASK	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	---	---	both
IPSET_ EXTENSIONEVT_ MSK	GCACT_ADDMSK	**gc_SetConfigData( )**	---	---	both
	GCACT_GET_MSK	**gc_SetConfigData( )**	---	---	both
	GCACT_SETMSK	**gc_SetConfigData( )**	---	---	both
	GCACT_SUBMSK	**gc_SetConfigData( )**	---	---	both
IPSET_FOIP	IPPARM_ T38_OFFERED	---	---	GCEV_OFFERED	both
	IPPARM_ T38_CONNECT	**gc_SetUserInfo( )**	---	---	both
	IPPARM_ T38_DISCONNECT	**gc_SetUserInfo( )**	---	---	both

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ H323_ RESPONSE_ CODE	IPPARM_BUSY_ CAUSE	**gc_SetConfigData( )**	---	---	H.323 only
IPSET_ IPPROTOCOL_ STATE	IPPARM_ CONTROL_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323 only
IPSET_ IPPROTOCOL_ STATE	IPPARM_ CONTROL_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323 only
	IPPARM_ SIGNALING_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323 only
	IPPARM_ SIGNALING_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323 only
IPSET_ LOCAL_ALIAS	IPPARM_ ADDRESS_ DOT_NOTATION	---	**gc_ReqService( )**	---	both
	IPPARM_ ADDRESS_EMAIL	---	**gc_ReqService( )**	---	both
	IPPARM_ ADDRESS_H323_ID	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ ADDRESS_PHONE	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ ADDRESS_ TRANSPARENT	---	**gc_ReqService( )**	GCEV_SERVICERESP	both
	IPPARM_ ADDRESS_URL	---	**gc_ReqService( )**	---	H.323 only
IPSET_ MEDIA_STATE	IPPARM_RX_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both
	IPPARM_RX_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both
	IPPARM_TX_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both
	IPPARM_TX_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323

IPSET_MIME
IPSET_MIME_ 200OK_TO_BYE
	IPPARM_MIME_PART	---	**gc_MakeCall( )**
			gc_SetInfo( )
			gc_CallAck( )
			gc_AcceptCall( )
			gc_AnswerCall( )
			gc_DropCall( )
			gc_Extension( )
				GCEV_OFFERED
				GCEV_PROCEEDING
				GCEV_ALERTING
				GCEV_CONNECTED
				GCEV_ DISCONNECTED
				GCEV_DROPCALL
				GCEV_TASKFAIL
				GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)
					SIP only

	IPPARM_MIME_ PART_BODY	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP only
	IPPARM_MIME_ PART_BODY_SIZE	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP only
	IPPARM_MIME_ PART_HEADER	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP only
	IPPARM_MIME_ PART_TYPE	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP only
IPSET_ MSG_H245	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	H.323 only
IPSET_ MSG_Q931	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	H.323 only
IPSET_ MSG_ REGISTRATION	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	both
IPSET_ MSG_SIP	IPPARM_MSG_SIP_ RESPONSE_CODE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	SIP only
	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	SIP only

IPSET_ NONSTANDARD CONTROL	IPPARM_ H221NON STANDARD	**gc_SetConfigData( )**
		gc_MakeCall( )
		gc_SetUserInfo( ) †
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

	IPPARM_ NONSTANDARD DATA_DATA	**gc_SetConfigData( )**
		gc_SetUserInfo( ) †
		gc_MakeCall( )
			gc_AnswerCall( )
			gc_MakeCall( )
			gc_DropCall( )
			gc_ReqService( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323

IPSET_ NONSTANDARD CONTROL	IPPARM_ NONSTANDARD DATA_OBJID	**gc_SetConfigData( )**
		gc_SetUserInfo( ) †
		gc_MakeCall( )
			gc_AnswerCall( )
			gc_MakeCall( )
			gc_DropCall( )
			gc_ReqService( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

IPSET_ NONSTANDARD DATA	IPPARM_ H221NON STANDARD	**gc_SetConfigData( )**
		gc_MakeCall( )
		gc_SetUserInfo( ) †
			gc_AnswerCall( )
			gc_MakeCall( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

	IPPARM_ NONSTANDARD DATA_DATA	**gc_SetConfigData( )**
		gc_SetUserInfo( ) †
		gc_MakeCall( )
			gc_AnswerCall( )
			gc_MakeCall( )
			gc_DropCall( )
			gc_ReqService( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

	IPPARM_ NONSTANDARD DATA_OBJID	**gc_SetConfigData( )**
		gc_SetUserInfo( ) †
		gc_MakeCall( )
			gc_AnswerCall( )
			gc_MakeCall( )
			gc_DropCall( )
			gc_ReqService( )
				gc_Extension( ) (IPEXTID_GETINFO)	H.323 only

IPSET_ PROTOCOL	IPPARM_ PROTOCOL_ BITMASK	**gc_SetConfigData( )**
		gc_SetUserInfo( ) †
		gc_MakeCall( )
			gc_ReqService( )
			gc_MakeCall( )
				---	both

IPSET_ REG_INFO	IPPARM_ OPERATION_ DEREGISTER	---	**gc_ReqService( )**	---	both
	IPPARM_ OPERATION_ REGISTER	---	**gc_ReqService( )**	---	both
	IPPARM_ REG_ADDRESS	---	**gc_ReqService( )**	---	both
	IPPARM_REG_ AUTOREFRESH	---	**gc_ReqService( )**	---	SIP only
	IPPARM_ REG_TYPE	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ REG_SERVICE_ID	---	---	Forwarded automatically in a GCEV_SERVICERESP event	SIP only
	IPPARM_ REG_STATUS	---	---	Forwarded automatically in a GCEV_SERVICERESP event	both
IPSET_RTP_ ADDRESS	IPPARM_LOCAL	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both
	IPPARM_REMOTE	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_SIP_ MSGINFO	IPPARM_ CALLID_HDR (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED or GCEV_EXTENSION	SIP only
	IPPARM_ CONTACT_DISPLAY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP only
	IPPARM_ CONTACT_URI (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only
	IPPARM_CONTENT_ DISPOSITION (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only
	IPPARM_CONTENT_ ENCODING (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only
	IPPARM_CONTENT_ LENGTH (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only
	IPPARM_CONTENT_ TYPE (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only
	IPPARM_ DIVERSION_URI (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP only
	IPPARM_EVENT_ HDR (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_Extension( )**	From event type GCEV_EXTENSION	SIP only
	IPPARM_EXPIRES_ HDR (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323

IPSET_SIP_ MSGINFO	IPPARM_FROM (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_InvokeXfer( )
			gc_Extension( )
				From event type GCEV_REQ_XFER or GCEV_EXTENSION	SIP only

	IPPARM_ FROM_DISPLAY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_MakeCall( )
			gc_Extension( )
				From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP only

	IPPARM_ REFERRED_BY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_MakeCall( )
			gc_InvokeXfer( )
			gc_Extension( )
				From event type GCEV_OFFERED, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only

	IPPARM_ REFER_TO (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_InvokeXfer( )
			gc_Extension( )
				None; parameter is set-only	SIP only

	IPPARM_ REPLACES (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_MakeCall( )
			gc_Extension( )
				From event type GCEV_OFFERED or GCEV_EXTENSION	SIP only

	IPPARM_ REQUEST_URI (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_MakeCall( )
			gc_Extension( )
				From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP only

	IPPARM_SIP_HDR	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_MakeCall( )
			gc_InvokeXfer( )
			gc_Extension( )
				From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP only

	IPPARM_TO (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_InvokeXfer( )
			gc_Extension( )
				From event type GCEV_REQ_XFER or GCEV_EXTENSION	SIP only

	IPPARM_ TO_DISPLAY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL)
		gc_Extension( )
			gc_MakeCall( )
			gc_Extension( )
				From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP only

IPSET_ SIP_REQUEST_ ERROR	IPPARM_SIP_DNS_ CONTINUE	---	---	From event type GCEV_EXTENSION	SIP only
	IPPARM_SIP_SVC_ UNAVAIL	---	---	From event type GCEV_EXTENSION	SIP only
IPSET_ SIP_ RESPONSE_ CODE	IPPARM_BUSY_ REASON	**gc_SetConfigData( )**	---	---	SIP only
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_DOT_ NOTATION	---	**gc_ReqService( )**	---	H.323 only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_EMAIL	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ ADDRESS_ H323_ID	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ ADDRESS_PHONE	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ ADDRESS_ TRANSPARENT	---	**gc_ReqService( )**	---	H.323 only
	IPPARM_ ADDRESS_URL	---	**gc_ReqService( )**	---	H.323 only
IPSET_ SWITCH_ CODEC	IPPARM_ACCEPT	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	both
	IPPARM_ AUDIO_INITIATE	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	both
	IPPARM_ AUDIO_ REQUESTED	---	---	GCEV_EXTENSION (IPEXTID_ CHANGE_MODE)	both
	IPPARM_READY	---	---	GCEV_EXTENSION (IPEXTID_ CHANGE_MODE)	both
	IPPARM_REJECT	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	both
	IPPARM_ T38_INITIATE	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	both
	IPPARM_ T38_REQUESTED	---	---	GCEV_EXTENSION (IPEXTID_ CHANGE_MODE)	both
IPSET_ TRANSACTION	IPPARM_ TRANSACTION_ID	---	---	**gc_Extension( )** (Any ext_id)	both
IPSET_ TUNNELED SIGNALMSG	IPPARM TUNNELEDSIGNAL MSG_ALTERNATEID	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323 only
	IPPARM TUNNELEDSIGNAL MSG_CONTENT	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323 only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 33. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ TUNNELED SIGNALMSG	IPPARM TUNNELEDSIGNAL MSG_NSDATA_DATA	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323 only
	IPPARM TUNNELEDSIGNAL MSG_NSDATA_ H221NS	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323 only
	IPPARM TUNNELEDSIGNAL MSG_OBJID	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323 only
	IPPARM TUNNELEDSIGNAL MSG_PROTOCOL_ OBJID	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323 only
IPSET_ VENDORINFO	IPPARM_ H221NONSTD	**gc_SetConfigData( )**	**gc_Extension( )** (IPEXTID_ SENDMSG)	**gc_Extension( )** (IPEXTID_GETINFO)	H.323 only
	IPPARM_ VENDOR_ PRODUCT_ID	**gc_SetConfigData( )**	**gc_Extension( )** (IPEXTID_ SENDMSG)	**gc_Extension( )** (IPEXTID_GETINFO)	H.323 only
	IPPARM_ VENDOR_ VERSION_ID	**gc_SetConfigData( )**	**gc_Extension( )** (IPEXTID_ SENDMSG)	**gc_Extension( )** (IPEXTID_GETINFO)	H.323 only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

8.2 Parameter Set Reference

This section contains reference information on the parameters in each parameter set used for IP telephony under Global Call. The table in each of the following subsections lists and describes the individual parameters associated with the parameter set as well as indicating the data type, size, and defined values for the parameters.

The parameter sets documented in this section include:

- GCSET_CALL_CONFIG
- IPSET_CALLINFO
- IPSET_CONFERENCE
- IPSET_CONFIG
- IPSET_DTMF
- IPSET_EXTENSIONEVT_MSK
- IPSET_FOIP

- IPSET_H323_RESPONSE_CODE
- IPSET_IPPROTOCOL_STATE
- IPSET_LOCAL_ALIAS
- IPSET_MEDIA_STATE
- IPSET_MSG_H245
- IPSET_MSG_Q931
- IPSET_MSG_REGISTRATION
- IPSET_MSG_SIP
- IPSET_NONSTANDARDCONTROL
- IPSET_NONSTANDARDDATA
- IPSET_PROTOCOL
- IPSET_REG_INFO
- IPSET_RTP_ADDRESS
- IPSET_SIP_MSGINFO
- IPSET_SIP_REQUEST_ERROR
- IPSET_SIP_RESPONSE_CODE
- IPSET_SUPPORTED_PREFIXES
- IPSET_SWITCH_CODEC
- IPSET_TRANSACTION
- IPSET_TUNNELEDSIGNALMSG
- IPSET_VENDORINFO

8.2.1 GCSET_CALL_CONFIG

Table 34 shows the parameter IDs in the GCSET_CALL_CONFIG parameter set that are relevant in an IP context.

Table 34. GCSET_CALL_CONFIG Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
GCPARM_CALLPROC	Type: enumeration Size: sizeof(char) Values: • GCCONTROL_APP - The application must use **gc_CallAck( )** to send the Proceeding message. This is the default. • GCCONTROL_TCCL - The stack sends the Proceeding message automatically.	Used to specify if the Proceeding message is sent under application control or automatically by the stack	both

8.2.2 IPSET_CALLINFO

Table 35 shows the parameter IDs in the IPSET_CALLINFO parameter set.

Table 35. IPSET_CALLINFO Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_BEARERCAP	Type: string Size: max. length = 255	Bearer Capability IE	H.323 only
IPPARM_CALLDURATION	Type: unsigned int Size: sizeof(unsigned int)	Duration of the call	H.323 only
IPPARM_CALLID	Type for SIP: string Size for SIP: max. length = MAX_IP_SIP_CALLID_LENGTH Type for H.323: array of octets Size for H.323: MAX_IP_H323_ CALLID_LENGTH If protocol is unknown, MAX_IP_ CALLID_LENGTH defines the maximum Call ID length for any supported protocol.	Globally unique identifier (Call ID) used by the underlying protocol to identify the call **Note:** When using SIP, direct manipulation of the Call ID message header via IPSET_SIP_MSGINFO / IPPARM_CALLID_HDR overrides any value provided via this parameter.	both
IPPARM_ CONNECTIONMETHOD	Type: enumeration Size: sizeof(char) Values: • IP_CONNECTIONMETHOD_ FASTSTART • IP_CONNECTIONMETHOD_ SLOWSTART	The connection method: Fast Start or Slow Start. See Section 4.2, “Using Fast Start and Slow Start Setup”, on page 104 for more information.	both
IPPARM_DISPLAY	Type: string Size: max. length = MAX_DISPLAY_LENGTH (82), null-terminated	Display information. This information can be used by a peer as additional address information.	both
IPPARM_FACILITY	Type: string Size: max. length = 255	Facility IE associated with SETUP, CONNECT, or FACILITY message. A Global Call Extension ID of EXTID_RECEIVEMSG applies when the IE is in an incoming FACILITY message.	H.323 only
IPPARM_H245TUNNELING	Type: enumeration Size: sizeof(char) Values: • IP_H245TUNNELING_ON • IP_H245TUNNELING_OFF	Specify if tunneling is on or off. See Section 4.19, “Enabling and Disabling H.245 Tunneling”, on page 206 for more information.	H.323 only
For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.			

Table 35. IPSET_CALLINFO Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_ MEDIAWAITFORCONNECT	Size: sizeof(char) Values: • 0 = FALSE • 1 = TRUE	MediaWaitForConnect field in SETUP message.	H.323 only
IPPARM_PHONELIST	Type: string Size: max. length = MAX_ADDRESS_LENGTH (128)	Phone numbers that can be retrieved at the remote end point. **Note:** When issuing a **gc_MakeCall( )**, this information can also be sent through the **numberstr** parameter. See Section 7.3.16, “gc_MakeCall( ) Variances for IP”, on page 311 for more information.	both
IPPARM_ PRESENTATION_IND	Type: enumeration Size: sizeof(char) Values: • IP_PRESENTATIONALLOWED • IP_PRESENTATION RESTRICTED	PresentationIndicator field in incoming and outgoing SETUP messages. An application may use this field to control whether the Caller ID is presented to the user.	H.323 only
IPPARM_PROGRESS_IND	Type: string Size: max. length = 255	Data contained in the Progress Indicator IE in incoming PROGRESS messages. **Note:** Extension events associated with PROGRESS messages are masked by default.Enable them with gc_SetUserInfo(IPSET_ EXTENSIONEVT_MSK, GCACT_SETMSK, EXTENSIOEVT_CALL_ PROGRESS)	H.323 only
IPPARM_USERUSER_INFO	Type: unsigned char[ ] Size: max size = MAX_USERUSER_INFO_ LENGTH (131)	User-to-user information	H.323 only
For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.			

8.2.3 IPSET_CONFERENCE

Table 36 shows the parameter IDs in the IPSET_CONFERENCE parameter set.

Table 36. IPSET_CONFERENCE Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_CONFERENCE_GOAL	Type: enumeration Size: sizeof(char) Values: • IP_CONFERENCEGOAL_UNDEFINED • IP_CONFERENCEGOAL_CREATE • IP_CONFERENCEGOAL_JOIN • IP_CONFERENCEGOAL_INVITE • IP_CONFERENCEGOAL_ CAP_NEGOTIATION • IP_CONFERENCEGOAL_ SUPPLEMENTARY_SRVC	The conference functionality to be achieved	H.323 only
IPPARM_CONFERENCE_ID	Type: string Size: max. length = IP_CONFERENCE_ID_LENGTH (16)	The conference identifier	H.323 only
1. For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1. 2. Conference ID retrieval is only relevant when an application is in a conference. In a peer-to-peer call, the conference ID does not signify a call identifier. The application should use IPPARM_CALLID to retrieve the call identifier. See Section 8.2.2, “IPSET_CALLINFO”, on page 358 for more information.			

8.2.4 IPSET_CONFIG

Table 37 shows the parameter IDs in the IPSET_CONFIG parameter set.

Table 37. IPSET_CONFIG Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_AUTHENTICATION_ CONFIGURE	Type: IP_ AUTHENTICATION Size: sizeof(IP_ AUTHENTICATION)	Used to add or modify a SIP authentication quadruplet. This parameter is only valid for the **gc_SetAuthenticationInfo( )** function.	SIP only
IPPARM_AUTHENTICATION_ REMOVE	Type: IP_ AUTHENTICATION Size: sizeof(IP_ AUTHENTICATION)	Used to remove a SIP authentication quadruplet based on the realm and identity strings in IP_AUTHENTICATION; the username and password. This parameter is only valid for the **gc_SetAuthenticationInfo( )** function.	SIP only
IPPARM_CONFIG_TOS	Type: char Size: sizeof(char)	Set the Type of Service (TOS) byte. Valid values are in the range 0 to 255. The default value is 0.	both

Table 37. IPSET_CONFIG Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_OPERATING_MODE	Type: int Size: sizeof(int)	Used when getting or setting the T.38 Fax Server mode. Possible values are: • IP_MANUAL_MODE	both
IPPARM_REGISTER_SIP_ HEADER	Type: string Size: max. length = IP_SIP_HDR_ MAXLEN (255)	Used to register the names of SIP message header fields that the application needs to retrieve from incoming messages	SIP only

8.2.5 IPSET_DTMF

Table 38 shows the parameter IDs in the IPSET_DTMF parameter set. This parameter set is used to set DTMF-related parameters for the notification, suppression or sending of DTMF digits.

Table 38. IPSET_DTMF Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ DTMF_ALPHANUMERIC	Type: IP_DTMF_DIGITS Size: sizeof(IP_DTMF_DIGITS)	Used when sending or receiving DTMF via UII alphanumeric messages. The parameter value contains an IP_DTMF_DIGITS structure that includes the digit string.	both
IPPARM_ DTMF_RFC2833_ PAYLOAD_TYPE	Type: unsigned char Size: sizeof(char) Values: 96 to 127	Used to specify the RFC2833 RTP payload type. The default value is IP_USE_STANDARD_PAYLOADTYPE (101).	both
IPPARM_ SUPPORT_DTMF_ BITMASK	Type: int Size: sizeof(int)	Used to specify a bitmask that defines which DTMF transmission methods are to be supported. Possible values are: • IP_DTMF_TYPE_ALPHANUMERIC † • IP_DTMF_TYPE_INBAND_RTP • IP_DTMF_TYPE_RFC_2833	both

8.2.6 IPSET_EXTENSIONEVT_MSK

This parameter set is used to enable or disable the events associated with unsolicited notification such as the detection of DTMF or a change of connection state in an underlying protocol. Table 39 shows the parameter IDs in the IPSET_EXTENSIONEVT_MSK parameter set.

Table 39. IPSET_EXTENSIONEVT_MSK Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
GCPARM_GET_MSK	Type: int Size: sizeof(int)	Retrieve the bitmask of enabled events	both
GCACT_SETMSK	Type: int Size: sizeof(int)	Set the bitmask of enabled events.	both
GCACT_ADDMSK	Type: int Size: sizeof(int)	Add to the bitmask of enabled events	both
GCACT_SUBMSK	Type: int Size: sizeof(int)	Remove from the bitmask of enabled events	both
Values that can be used to make up the bitmask are: • EXTENSIONEVT_DTMF_ALPHANUMERIC (0x04) † • EXTENSIONEVT_SIGNALING_STATUS (0x08) • EXTENSIONEVT_STREAMING_STATUS (0x10) • EXTENSIONEVT_T38_STATUS (0x20)			

8.2.7 IPSET_FOIP

Table 40 shows the parameter IDs in the IPSET_FOIP parameter set.

Table 40. IPSET_FOIP Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_T38_OFFERED	Type: IP_CONNECT Size: sizeof(IP_CONNECT)	Used in a GC_PARM_BLK associated with an GCEV_OFFERED event to indicate that a T.38 session is requested.	both
IPPARM_T38_CONNECT	Type: IP_CONNECT Size: sizeof(IP_CONNECT)	Used when associating a T.38 Fax device with a Media device when switching from an audio session to a fax session.	both
IPPARM_T38_DISCONNECT	Type: IP_CONNECT Size: sizeof(IP_CONNECT)	Used when disassociating a T.38 Fax device with a Media device when switching from a fax session to an audio session.	both

8.2.8 IPSET_H323_RESPONSE_CODE

This parameter set is used to set the busy cause code that is used in the failure message sent when the local system is unable to accept additional incoming sessions.

Table 41. IPSET_H323_RESPONSE_CODE Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_ BUSY_CAUSE	Type: eIP_EC_TYPE Size: sizeof(int)	Used in a GC_PARM_BLK to specify the cause code to send when no additional incoming sessions can be accepted. Values: • IPEC_Q931Cause34NoCircuitChannelAvailable • IPEC_Q931Cause47ResourceUnavailableUnspecified	H.323 only

8.2.9 IPSET_IPPROTOCOL_STATE

This parameter set is used when retrieving notification of protocol signaling states via GCEV_EXTENSION events. Table 42 shows the parameter IDs in the IPSET_IPPROTOCOL_STATE parameter set.

Table 42. IPSET_IPPROTOCOL_STATE Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ SIGNALING_CONNECTED	Type: int Size: sizeof(int)	Call signaling for the call has been established with the remote endpoint	H.323 only
IPPARM_ SIGNALING_DISCONNECTED	Type: int Size: sizeof(int)	Call signaling for the call has been terminated	H.323 only
IPPARM_ CONTROL_CONNECTED	Type: int Size: sizeof(int)	Media control signaling for the call has been established with the remote endpoint	H.323 only
IPPARM_ CONTROL_DISCONNECTED	Type: int Size: sizeof(int)	Media control signaling for the call has been terminated	H.323 only

8.2.10 IPSET_LOCAL_ALIAS

Table 43 shows the parameter IDs in the IPSET_LOCAL_ALIAS parameter set.

Table 43. IPSET_LOCAL_ALIAS Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ ADDRESS_DOT_NOTATION	Type: string Size: max. length = 255	A valid IP address	both
IPPARM_ ADDRESS_EMAIL	Type: string Size: max. length = 255	e-mail address composed of characters from the set “[A-Z][a-z][0-9]_-.@”	both
IPPARM_ ADDRESS_H323_ID	Type: string Size: max. length = 255	A valid H.323 ID	H.323 only
IPPARM_ ADDRESS_PHONE	Type: string Size: max. length = 255	An E.164 telephone number	H.323 only
IPPARM_ ADDRESS_TRANSPARENT	Type: string Size: max. length = 255	Unspecified address type	both
IPPARM_ ADDRESS_URL	Type: string Size: max. length = 255	A valid URL composed of characters from the set “[A-Z][a-z][0-9]-.”. Must contain at least one “.” and may not begin or end with a “-”.	H.323 only

Note: For SIP, IPPARM_LOCAL_ALIAS is not used for the alias (or Address of Record), but is used for the transport address or contact.

8.2.11 IPSET_MEDIA_STATE

Table 44 shows the parameter IDs in the IPSET_MEDIA_STATE parameter set.

Table 44. IPSET_MEDIA_STATE Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ RX_CONNECTED	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming has been initiated in the receive direction from the remote endpoint. The IP_CAPABILITY structure includes coder information negotiated with the remote peer. See Section 4.17, “Enabling and Disabling Unsolicited Notification Events”, on page 205 for more information.	both
IPPARM_ RX_DISCONNECTED	Type: int Size: sizeof(int)	Streaming in the receive direction from the remote endpoint has been terminated.	both

Table 44. IPSET_MEDIA_STATE Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ TX_CONNECTED	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming has been initiated in the transmit direction toward the remote endpoint. The IP_CAPABILITY structure includes coder information negotiated with the remote peer. See Section 4.17, "Enabling and Disabling Unsolicited Notification Events", on page 205 for more information.	both
IPPARM_ TX_DISCONNECTED	Type: int Size: sizeof(int)	Streaming in the transmit direction toward the remote endpoint has been terminated.	both

8.2.12 IPSET_MIME and IPSET_MIME_200OK_TO_BYE

Table 45 shows the parameter IDs in the IPSET_MIME and IPSET_MIME_200OK_TO_BYE parameter sets which are used when sending and receiving MIME-encoded SIP messages. The same parameters apply to both parameter sets. When using the IPSET_MIME_200OK_TO_BYE parameter set ID, that same set ID must be used in all data blocks associated with the message.

Table 45. IPSET_MIME and IPSET_MIME_200OK_TO_BYE Parameter Sets

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_MIME_PART	Type: pointer to GC_PARM_BLK Size: 4 bytes	Required parameter. Used to set or get SIP message MIME part(s). Parameter value is a pointer to a GC_PARM_BLK structure that contains a list of pointers to one or more GC_PARM_BLK structures that contain MIME message parts.	SIP only
IPPARM_ MIME_PART_BODY	Type: char * Size: 4 bytes	Required parameter. Used to copy MIME part body between application and Global Call space. Parameter value is a pointer to a MIME part body.	SIP only
IPPARM_ MIME_PART_BODY_SIZE	Type: Unsigned int Size: 4 bytes	Required parameter. Used to indicate the actual size of the MIME part body, not including MIME part headers.	SIP only
IPPARM_ MIME_PART_HEADER	Type: Null-terminated string Size: IP_MIME_PART_ HEADER_MAXLEN (=255)	Optional parameter. Used to contain MIME part header field in format of "field-name: field-value". Field-name can be any string other than "Content-type". Content is not checked by Global Call before insertion into SIP message.	SIP only
IPPARM_ MIME_PART_TYPE	Type: Null-terminated string Size: IP_MIME_PART_ HEADER_MAXLEN (=255)	Required parameter. Used to contain name and value of the MIME part content type field. String must begin with the field name "Content-Type:".	SIP only

8.2.13 IPSET_MSG_H245

Table 46 shows the parameter IDs in the IPSET_MSG_H245 parameter set. This parameter set is used with the **gc_Extension( )** and the IPEXTID_SENDMSG extension and encapsulates all the parameters required to send an H.245 message.

Table 46. IPSET_MSG_H245 Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Possible values for H.245 messages are: • IP_MSGTYPE_H245_INDICATION	H.323 only

8.2.14 IPSET_MSG_Q931

Table 47 shows the parameter IDs in the IPSET_MSG_Q931 parameter set. This parameter set is used with the **gc_Extension( )** and the IPEXTID_SENDMSG extension and encapsulates all the parameters required to send or receive a Q.931 message.

Table 47. IPSET_MSG_Q931 Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Possible values for Q.931 messages are: • IP_MSGTYPE_Q931_FACILITY • IP_MSGTYPE_Q931_PROGRESS	H.323 only

8.2.15 IPSET_MSG_REGISTRATION

Table 48 shows the parameter IDs in the IPSET_MSG_REGISTRATION parameter set. This parameter set is used with the **gc_Extension( )** and the IPEXTID_SENDMSG extension and encapsulates all the parameters required to send a registration message. For information on the use of this parameter set, see Section 4.16.3, “Nonstandard Registration Message”, on page 203.

Table 48. IPSET_MSG_REGISTRATION Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Possible value for registration messages is: • IP_MSGTYPE_REG_NONSTD	both

8.2.16 IPSET_MSG_SIP

Table 49 shows the parameter IDs in the IPSET_MSG_SIP parameter set. This parameter set is used to the response code or message type in an outgoing SIP message that is sent using **gc_Extension( )** and the IPEXTID_SENDMSG extension. It is also used to identify the type of SIP event that is contained in a supported or registered SIP message that is passed to the application as a GCEV_EXTENSION event or as a GCEV_CALLINFO event for SIP INFO messages only.

Table 49. IPSET_MSG_SIP Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSG_SIP_ RESPONSE_CODE	Type: int Size: sizeof(int)	Used to set the numerical response code to send in a SIP response message, or to extract the code from a received response message.	SIP only
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Type of supported SIP message to send or SIP event type to retrieve from an extension event. Defined values are: • IP_MSGTYPE_SIP_INFO • IP_MSGTYPE_SIP_INFO_FAILED • IP_MSGTYPE_SIP_INFO_OK • IP_MSGTYPE_SIP_NOTIFY • IP_MSGTYPE_SIP_NOTIFY_ACCEPT • IP_MSGTYPE_SIP_NOTIFY_REJECT • IP_MSGTYPE_SIP_OPTIONS • IP_MSGTYPE_SIP_OPTIONS_FAILED • IP_MSGTYPE_SIP_OPTIONS_OK • IP_MSGTYPE_SIP_SUBSCRIBE • IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT • IP_MSGTYPE_SIP_SUBSCRIBE_EXPIRE (receive only) • IP_MSGTYPE_SIP_SUBSCRIBE_REJECT	SIP only

8.2.17 IPSET_NONSTANDARDCONTROL

Table 50 shows the parameter IDs in the IPSET_NONSTANDARDCONTROL parameter set.

Table 50. IPSET_NONSTANDARDCONTROL Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ NONSTANDARDDATA_DATA	Type: string Size: max. length = MAX_NS_PARM_ DATA_LENGTH (128)	Contains the nonstandard data supplied, if any. If nonstandard data was not supplied, this parameter should not be present in the parm block.	H.323 only
IPPARM_ NONSTANDARDDATA_OBJID	Type: Uint[ ] Size: max. length = MAX_NS_PARM_ OBJID_LENGTH (40)	Contains the nonstandard object ID supplied, if any. If a nonstandard object ID was not provided, this parameter should not be present in the parm block.	H.323 only
IPPARM_H221NONSTANDARD	Type: IP_H221NONSTANDARD Size: sizeof(IP_H221NONSTANDARD)	Contains an H.221 nonstandard data identifier.	H.323 only
For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.			

8.2.18 IPSET_NONSTANDARDDATA

Table 51 shows the parameter IDs in the IPSET_NONSTANDARDDATA parameter set.

Table 51. IPSET_NONSTANDARDDATA Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ NONSTANDARDDATA_DATA	Type: string Size: max. length = MAX_NS_PARM_DATA_ LENGTH (128)	Contains the nonstandard data supplied, if any. If nonstandard data was not supplied, this parameter should not be present in the parm block.	H.323 only
IPPARM_ NONSTANDARDDATA_OBJID	Type: Uint[ ] Size: max. length = MAX_NS_PARM_OBJID_ LENGTH (40)	Contains the nonstandard object ID supplied, if any. If a nonstandard object ID was not provided, this parameter should not be present in the parm block.	H.323 only
IPPARM_H221NONSTANDARD	Type: IP_H221NONSTANDARD Size: sizeof(IP_H221NONSTANDARD)	Contains an H.221 nonstandard data identifier.	H.323 only
For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.			

8.2.19 IPSET_PROTOCOL

Table 52 shows the parameter IDs in the IPSET_PROTOCOL parameter set.

Table 52. IPSET_PROTOCOL Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_PROTOCOL_BITMASK	Type: char Size: sizeof(char)	The IP protocol to use. Defined values (which may be OR'ed) are: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP	both

8.2.20 IPSET_REG_INFO

Table 53 shows the parameter IDs in the IPSET_REG_INFO parameter set.

Table 53. IPSET_REG_INFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_OPERATION_ REGISTER	Type: char Size: sizeof(char)	Used to manipulate registration information when registering an endpoint with a gatekeeper/registrar. Possible values are: • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE • IP_REG_QUERY_INFO (SIP only) • IP_REG_SET_INFO	both
IPPARM_OPERATION_ DEREGISTER	Type: char Size: sizeof(char)	Used when deregistering an endpoint with a gatekeeper/registrar. Possible values are: • IP_REG_DELETE_ALL – Discard the registration data in the local database • IP_REG_MAINTAIN_LOCAL_INFO – Keep the registration data in the local database	both
IPPARM_REG_ADDRESS	Type: IP_REGISTER_ ADDRESS Size: sizeof(IP_REGISTER_ ADDRESS)	Address information to be registered with a gatekeeper/registrar. See the reference page for IP_REGISTER_ADDRESS on page 392 for details.	both
IPPARM_REG_ AUTOREFRESH	Type: char Size: sizeof(char)	Used to enable/disable autorefresh of SIP registration bindings. Possible values are: • IP_REG_AUTOREFRESH_DISABLE • IP_REG_AUTOREFRESH_ENABLE Default behavior if this parameter is not specified is to autorefresh bindings.	SIP only

Table 53. IPSET_REG_INFO Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_REG_TYPE	Type: int Size: sizeof(int)	The registration type. Possible values are: • IP_REG_GATEWAY • IP_REG_TERMINAL	H.323 only
IPPARM_REG_SERVICEID	Type: int Size: sizeof(int)	The Service ID that was handed back to the application when it initiate the registration	SIP only
IPPARM_REG_STATUS	Type: char Size: sizeof(char)	Provides an indication of whether the endpoint registration with a gatekeeper/registrar was successful or not. Possible values are: • IP_REG_CONFIRMED • IP_REG_REJECTED	both

8.2.21 IPSET_RTP_ADDRESS

Table 53 shows the parameter IDs in the IPSET_RTP_ADDRESS parameter set.

Table 54. IPSET_RTP_ADDRESS Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_LOCAL	Type: int Size: sizeof(int)	Used when retrieving RTP address of the local endpoint of an RTP stream as contained in a connection event.	both
IPPARM_REMOTE	Type: int Size: sizeof(int)	Used when retrieving RTP address of the remote endpoint of an RTP stream as contained in a connection event.	both

8.2.22 IPSET_SIP_MSGINFO

Table 55 shows the parameter IDs in the IPSET_SIP_MSGINFO parameter set. Note that access to SIP message header info fields is disabled by default and must be explicitly enabled by setting the IP_SIP_MSGINFO_ENABLE mask value in the sip_msginfo_mask field of the IP_VIRTBOARD structure before starting the virtual board.

Notes: *1.* All parameter IDs in this parameter set are deprecated except IPPARM_SIP_HDR. The deprecated parameter IDs will remain in the IP Call Control Library for backward compatibility, but there will be no further development in relation to these parameter IDs.

2. All of the MAXLEN defines for the deprecated SIP header fields are equated to 255 bytes.

3. The maximum data length for the IPPARM_SIP_HDR parameter ID is not limited to 255 bytes. Applications using this parameter ID **must** use the "extended" **gc_util_...** utility functions, which are capable of handling parameter data longer than 255 bytes.

Table 55. IPSET_SIP_MSGINFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_CALLID_HDR (deprecated)	Type: string Max size: IP_CALLID_HDR_ MAXLEN	Deprecated parameter to set or retrieve the globally unique call identifier (Call-ID header field) in SIP messages. **Note:** Any value set via this parameter overrides any Call-ID value set via IPSET_CALLINFO / IPPARM_CALLID.	SIP only
IPPARM_CONTACT_DISPLAY (deprecated)	Type: string Max size: IP_CONTACT_ DISPLAY_MAXLEN	Deprecated parameter to set or retrieve display name in Contact header field of SIP messages	SIP only
IPPARM_CONTACT_URI (deprecated)	Type: string Max Size: IP_CONTACT_ URI_MAXLEN	Deprecated parameter to set or retrieve URI in Contact header field of SIP messages	SIP only
IPPARM_CONTENT_ DISPOSITION (deprecated)	Type: string Max Size: IP_CONTENT_ DISPOSITION_MAXLEN	Deprecated parameter to set or retrieve Content-Disposition header field of SIP messages	SIP only
IPPARM_CONTENT_ ENCODING (deprecated)	Type: string Max Size: IP_CONTENT_ ENCODING_MAXLEN	Deprecated parameter to set or retrieve Content-Encoding header field of SIP messages	SIP only
IPPARM_CONTENT_LENGTH (deprecated)	Type: string Max Size: IP_CONTENT_ LENGTH_MAXLEN	Deprecated parameter to set or retrieve Content-Length header field of SIP messages	SIP only
IPPARM_CONTENT_TYPE (deprecated)	Type: string Max Size: IP_CONTENT_ TYPE_MAXLEN	Deprecated parameter to set or retrieve Content-Type header field of SIP messages	SIP only
IPPARM_DIVERSION_URI (deprecated)	Type: string Max Size: IP_DIVERSION_ URI_MAXLEN	Deprecated parameter to set or retrieve URI in the Diversion header field of SIP messages	SIP only

Table 55. IPSET_SIP_MSGINFO Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_EVENT_HDR (deprecated)	Type: string Max Size: IP_EVENT_HDR_ MAXLEN	Deprecated parameter to set or retrieve Event header field of SIP messages	SIP only
IPPARM_EXPIRES_HDR (deprecated)	Type: string Max Size: IP_EXPIRES_ HDR_TYPE_MAXLEN	Deprecated parameter to set or retrieve Expires header field of SIP messages	SIP only
IPPARM_FROM (deprecated)	Type: string Max Size: IP_FROM_ MAXLEN	Deprecated parameter to set or retrieve complete From header field (display name, URI, parameters) of SIP messages	SIP only
IPPARM_FROM_DISPLAY (deprecated)	Type: string Max Size: IP_FROM_ DISPLAY_MAXLEN	Deprecated parameter to set or retrieve display name in the From header field of SIP messages	SIP only
IPPARM_REFER_TO (deprecated)	Type: string Max Size: IP_REFER_TO_ MAXLEN	Deprecated parameter to set Refer-To header field in SIP REFER messages in call transfer operations (set only)	SIP only
IPPARM_REFERRED_BY (deprecated)	Type: string Max Size: IP_REFERRED_ BY_MAXLEN	Deprecated parameter to set or retrieve Referred-By header field in SIP messages	SIP only
IPPARM_REPLACES (deprecated)	Type: string Max Size: IP_REPLACES_ MAXLEN	Deprecated parameter to set or retrieve Replaces parameter in Refer-To header of SIP REFER messages (attended call transfer only)	SIP only
IPPARM_REQUEST_URI (deprecated)	Type: string Max Size: IP_REQUEST_ URI_MAXLEN	Deprecated parameter to set Request-URI of SIP messages	SIP only
IPPARM_SIP_HDR	Type: string Max Size: IP_CFG_PARM_ DATA_MAXLEN	Used to set or retrieve standard or proprietary header fields in SIP messages	SIP only
IPPARM_TO_DISPLAY (deprecated)	Type: string Max Size: IP_TO_DISPLAY_ MAXLEN	Deprecated parameter to set or retrieve display name in the To header field of SIP messages	SIP only
IPPARM_TO (deprecated)	Type: string Max Size: IP_TO_MAXLEN	Deprecated parameter to set or retrieve complete To header field (display name, URI, parameters) of SIP messages	SIP only

8.2.23 IPSET_SIP_REQUEST_ERROR

This parameter set is used to indicate that a SIP request has had a transport failure. busy cause code that is used in the failure message sent when the local system is unable to accept additional incoming SIP sessions.

Table 56. IPSET_SIP_REQUEST_ERROR Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_SIP_ DNS_CONTINUE	Type: REQUEST_ ERROR Size: sizeof(REQUEST_ ERROR)	Used in a GCEV_EXTENSION event to indicate that a SIP request had a transport failure and is being retried using address information from the DNS server. The REQUEST_ERROR structure contains an Error field with one of following parameter values to indicate the cause of the transport failure: • IP_SIP_503_RCVD (503 Service Unavailable response received) • IP_SIP_FAILED (general transport error) • IP_SIP_NETWORK_ERROR (network error or local failure) • IP_SIP_TIMEOUT (timeout before response received)	SIP only
IPPARM_SIP_ SVC_UNAVAIL	Type: REQUEST_ ERROR Size: sizeof(REQUEST_ ERROR)	Used in a GCEV_EXTENSION event to indicate that a SIP request had a fatal transport failure.The REQUEST_ERROR structure contains an Error field with one of following parameter values to indicate the cause of the transport failure: • IP_SIP_503_RCVD (X503 Service Unavailable response received) • IP_SIP_FAILED (general transport error) • IP_SIP_NETWORK_ERROR (network error or local failure) • IP_SIP_RETRY_FAILED (retry logic error; no retry attempted) • IP_SIP_TIMEOUT (timeout before response received)	SIP only

8.2.24 IPSET_SIP_RESPONSE_CODE

This parameter set is used to set the busy cause code that is used in the failure message sent when the local system is unable to accept additional incoming SIP sessions.

Table 57. IPSET_SIP_RESPONSE_CODE Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_ BUSY_REASON	Type: eIP_EC_TYPE Size: sizeof(int)	Used in a GC_PARM_BLK to specify the cause code to send when no additional incoming sessions can be accepted. Values: • IPEC_SIPReasonStatus480TemporarilyUnavailable • IPEC_SIPReasonStatus486BusyHere • IPEC_SIPReasonStatus600BusyEverywhere	SIP only

8.2.25 IPSET_SUPPORTED_PREFIXES

Table 58 shows the parameter IDs in the IPSET_SUPPORTED_PREFIXES parameter set.

Table 58. IPSET_SUPPORTED_PREFIXES Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ ADDRESS_DOT_NOTATION	Type: string Size: max. length = 255	A valid IP address in dot notation	H.323 only
IPPARM_ ADDRESS_EMAIL	Type: string Size: max. length = 255	An e-mail address composed of characters from the set “[A-Z][a-z][0-9]_-.@”	H.323 only
IPPARM_ ADDRESS_H323_ID	Type: string Size: max. length = 255	A valid H.323 ID	H.323 only
IPPARM_ ADDRESS_PHONE	Type: string Size: max. length = 255	An E.164 telephone number	H.323 only
IPPARM_ ADDRESS_TRANSPARENT	Type: string Size: max. length = 255	Unspecified address type	H.323 only
IPPARM_ADDRESS_URL	Type: string Size: max. length = 255	A valid URL composed of characters from the set “[A-Z][a-z][0-9]-.”. Must contain at least one “.” and may not begin or end with a “-”.	H.323 only

8.2.26 IPSET_SWITCH_CODEC

Table 59 shows the parameter IDs in the IPSET_SWITCH_CODEC parameter set.

Table 59. IPSET_SWITCH_CODEC Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ACCEPT	Type: int Size: sizeof(int)	Used to accept an incoming coder switch request.	both
IPPARM_AUDIO_INITIATE	Type: int Size: sizeof(int)	Used to initiate the sending of a RequestMode (H.323) or REINVITE (SIP) message to the remote side to switch from T.38 fax to audio.	both
IPPARM_AUDIO_REQUESTED	Type: int Size: sizeof(int)	Provides notification of an incoming request to switch from T.38 fax to audio.	both
IPPARM_READY	Type: int Size: sizeof(int)	Provides notification that the media is ready.	both
IPPARM_REJECT	Type: int Size: sizeof(int)	Used to reject an incoming request to switch from audio to T.38 fax or vice versa.	both
IPPARM_T38_INITIATE	Type: int Size: sizeof(int)	Used to initiate the sending of a RequestMode (H.323) or REINVITE (SIP) message to the remote side to switch from audio to T.38 fax.	both
IPPARM_T38_REQUESTED	Type: int Size: sizeof(int)	Provides notification of an incoming request to switch from audio to T.38 fax.	both

8.2.27 IPSET_TRANSACTION

Table 60 shows the parameter IDs in the IPSET_TRANSACTION parameter set.

Table 60. IPSET_TRANSACTION Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_TRANSACTION_ID	Type: int Size: sizeof(int)	Used to uniquely identify any transaction	H.323 only

8.2.28 IPSET_TUNNELEDSIGNALMSG

Table 61 shows the parameter IDs in the IPSET_TUNNELEDSIGNALMSG parameter set, which is used when sending or receiving tunneled signaling messages (TSMs) in the H.323 protocol.

Table 61. IPSET_TUNNELEDSIGNALMSG Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ TUNNELEDSIGNALMSG_ ALTERNATEID	Type: IP_TUNNEL PROTOCOL_ ALTID Size: sizeof(IP_TUNNEL PROTOCOL_ ALTID)	Used to contain a tunneled protocol alternate identifier in a tunneled signaling message (TSM). Either this or the tunneled protocol object ID can exist in a TSM. If the application is using a tunneled protocol object ID when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323 only
IPPARM_ TUNNELEDSIGNALMSG_ CONTENT	Type: string Max length: MAX_IE_ LENGTH (255)	Used to contain any data content of a tunneled signaling message (TSM), which is a sequence of octet strings.	H.323 only
IPPARM_ TUNNELEDSIGNALMSG_ NSDATA_DATA	Type: string Max length: MAX_NS_ PARAM_DATA_ LENGTH (255)	Used to contain any non-standard data in a tunneled signaling message (TSM). If no non-standard data is being sent in a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323 only
IPPARM_ TUNNELEDSIGNALMSG_ NSDATA_H221NS	Type: IP_H221 NONSTANDARD Size: sizeof(IP_H221NON STANDARD)	Used to contain an H.221 non-standard data identifier in a tunneled signaling message (TSM). Either this ID or the non-standard data object ID can exist in a TSM's non-standard data. If non-standard data is not being sent, or if a non-standard data object ID is being used when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323 only
IPPARM_ TUNNELEDSIGNALMSG_ NSDATA_OBJID	Type: string Max length: MAX_NS_ PARAM_OBJID_ LENGTH (40)	Used to contain a non-standard data object identifier in a tunneled signaling message (TSM). Either this ID or an H.221 non-standard data ID can exist in a TSM's non-standard data. If non-standard data is not being sent, or if an H.221 non-standard data ID is being used when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323 only
IPPARM_ TUNNELEDSIGNALMSG_ PROTOCOL_OBJID	Type: string Max length: MAX_TSM_ POID_PARAM_ LENGTH (128)	Used to contain a tunneled protocol object identifier in a tunneled signaling message (TSM). Either this or the tunneled protocol alternate ID can exist in a TSM. If the application is using an alternate identifier when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323 only

8.2.29 IPSET_VENDORINFO

Table 62 shows the parameter IDs in the IPSET_VENDORINFO parameter set.

Table 62. IPSET_VENDORINFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_H221NONSTD	Type: IP_H221NONSTANDARD Size: sizeof(IP_ H221NONSTANDARD)	Contains country code, extension code and manufacturer code. See the reference page for IP_H221NONSTANDARD on page 391 for details.	H.323 only
IPPARM_ VENDOR_PRODUCT_ID	Type: string Size: max. length = MAX_PRODUCT_ID_LENGTH (32)	Vendor product identifier	H.323 only
IPPARM_ VENDOR_VERSION_ID	Type: string Size: max. length = MAX_VERSION_ID_LENGTH (32)	Vendor version identifier	H.323 only
For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.			

IP-Specific Data Structures 9

This chapter describes the data structures that are specific to IP technology.

Note: These data structures are defined in the *gcip.h* header file.

GC_PARM_DATA_EXT

```
typedef struct
{
    unsigned long      version;
    void*              pInternal;
    unsigned long      set_ID;
    unsigned long      parm_ID;
    unsigned long      data_size;
    void*              pData;
}GC_PARM_DATA_EXT, *GC_PARM_DATA_EXTP;
```

■ Description

The GC_PARM_DATA_EXT structure contains parameter data retrieved from a GC_PARM_BLK by the **gc_util_find_parm_ex( )** and **gc_util_next_parm_ex( )** functions. These functions were added to the Global Call API library to support the retrieval of parameters that may exceed 255 bytes in length, but these fucntions always return the retrieved parameter information in a GC_PARM_DATA_EX structure regardless of whether the parameter data exceeds 255 bytes.

The set ID and parm ID as a pair identify the parameter. Set IDs and parm IDs that are common to multiple Global Call technologies are listed in the *Global Call API Library Reference*, and additional technology-specific parameters are listed in each of the various Global Call Technology Guides. Unless a particular set ID/parm IP pair specifically indicates that it supports parameter data that exceeds 255 bytes in length, users should assume that the parameter data length does not exceed 255.

Note: The only Global Call parameter that currently supports data longer than 255 bytes is IPSET_SIP_MSGINFO / IPPARM_SIP_HDR, which is used for SIP message headers.

Applications should use the **INIT_GC_PARM_DATA_EXT( )** function to initialize the structure with the correct version number and default field values before setting the appropriate values.

■ Field Descriptions

The fields of GC_PARM_DATA_EXT are described as follows:

version
: identifies the version of the data structure implementation. This field is reserved for library use and should **not** be modified by applications.

pInternal
: pointer used to identify position within the GC_PARM_BLK structure. This field is reserved for library use and should **not** be used or modified by applications.

set_id
: the set ID of the retrieved parameter

parm_id
: the parameter ID of the retrieved parameter

data_size
: the size of the retrieved parameter data in bytes

pData

pointer to the first byte of the parameter value buffer.

IP_ADDR

```
typedef struct
{
    unsigned char     ip_ver;
    union
    {
        unsigned int      ipv4;
        unsigned int      ipv6[4]
    }u_ipaddr;
}IP_ADDR, *IP_ADDRP;
```

Description

The IP_ADDR structure is used to specify a local IP address.

Field Descriptions

The fields of the IP_ADDR data structure are described as follows:

ip_ver

The version of the local IP address. Possible values are:

- IPVER4

u_ipaddr

A union that contains the actual address. The datatype is different depending on whether the address is an IPv4 or an IPv6 address.

Note: IPv6 addresses are not currently supported.

For an IPv4 address, the address must be stored in memory using the network byte order (big endian) rather than the little-endian byte order of the Intel architecture. A socket API, **htonl( )**, is available to convert from host byte order to network byte order. As an example, to specify an IP address of 127.10.20.30, you may use either of the following C statements:

`ipv4 = 0x1e140a7f` -or-

`ipv4 = htonl(0x7f0a141e)`

For more information on the byte order of IPv4 addresses, see RFC 791 and RFC 792.

IP_AUDIO_CAPABILITY

```
typedef struct
{
    unsigned long   frames_per_pkt;
    long            VAD;
} IP_AUDIO_CAPABILITY;
```

Description

The IP_AUDIO_CAPABILITY data structure is used to allow some minimum set of information to be exchanged together with the audio codec identifier.

Field Descriptions

The fields of the IP_AUDIO_CAPABILITY data structure are described as follows:

frames_per_pkt

When bundling more than one audio frame into a single transport packet, this value should represent the maximum number of frames per packet that will be sent on the wire. When set to zero, indicates that the exact number of frames per packet is not known, or that the data is not applicable. This field can also be set to GCCAP_dontCare to indicate that any supported value is valid.

Note: For G.711 coders, this field represents the frame size (for example, 10 msec); the frames per packet value is fixed at 1 fpp. For other coders, this field represents the frames per packet and the frame size is fixed. See Section 4.3.2, “Setting Coder Information”, on page 108 for more information.

VAD

Identifies whether voice activated detection (VAD) is enabled or disabled. Possible values are:

- GCPV_ENABLE – VAD enabled
- GCPV_DISABLE – VAD disabled
- GCCAP_dontCare – Any supported value is valid

IP_AUTHENTICATION

```
typedef struct
{
    unsigned short  version;
    char*           realm;
    char*           identity;
    char*           username;
    char*           password;
} IP_AUTHENTICATION;
```

■ Description

The IP_AUTHENTICATION data structure is used when setting or removing SIP authentication quadruplets.

Applications should use the **INIT_IP_AUTHENTICATION( )** function to initialize the structure with the correct version number and void pointers for each of the strings before setting the appropriate values.

■ Field Descriptions

The fields of the IP_AUTHENTICATION data structure are described as follows:

version
: The version number of the data structure. The correct value is set by the **INIT_IP_AUTHENTICATION( )** initialization function and should not be overridden.

realm
: A null-terminated string that defines the protected domain. This string is case-insensitive and must always be supplied.

identity
: A null-terminated string that allows applications to optionally specify different username/ password pairs for different identities in the same realm. The identity is a URI and must conform to URI syntax, including starting with the scheme (namely “sip:” or “sips:”). If only one username and password applies to a given realm or if setting a default username and password for a multi-identity realm, use an empty string ("") for this field. This field is case-insensitive.

username
: A null-terminated string providing the user’s name in the specified realm. This field is case-sensitive. This field must always contain a non-empty string when the structure is associated with an IPPARM_AUTHENTICATION_CONFIGURE parameter. This field is ignored when the structure is associated with an IPPARM_AUTHENTICATION_REMOVE parameter.

password
: A null-terminated string providing password associated with the user’s name in the specified realm. This field is case-sensitive. This field is ignored when the structure is associated with an IPPARM_AUTHENTICATION_REMOVE parameter.

IP_CAPABILITY

```
typedef struct
{
    int                     capability;
    int                     type;
    int                     direction;
    int                     payload_type;
    IP_CAPABILITY_UNION     extra;
    char                    rfu[0x10];
} IP_CAPABILITY;
```

Description

The IP_CAPABILITY data structure provides a level of capability information in addition to simply the capability or codec identifier.

Note: The IP_CAPABILITY data structure is not intended to provide all the flexibility of the H.245 terminal capability structure, but provides a first level of useful information in addition to the capability or codec identifier.

Field Descriptions

The fields of the IP_CAPABILITY data structure are described as follows:

capability

The IP Media capability for this structure. Possible values are:

- GCCAP_AUDIO_g711Alaw64k
- GCCAP_AUDIO_g711Ulaw64k
- GCCAP_AUDIO_g7231_5_3k
- GCCAP_AUDIO_g7231_6_3k
- GCCAP_AUDIO_g729AnnexA
- GCCAP_AUDIO_g729AnnexAwAnnexB
- GCCAP_AUDIO_NO_AUDIO
- GCCAP_DATA_t38UDPFax
- GCCAP_dontCare

type

The category of capability specified in this structure. Indicates which member of the IP_CAPABILITY_UNION union is being used. Possible values are:

- GCCAPTYPE_AUDIO – Audio
- GCCAPTYPE_RDATA – Data

direction

The capability direction code for this capability. Possible values are:

- IP_CAP_DIR_LCLTRANSMIT – Indicates a transmit capability for the local endpoint.
- IP_CAP_DIR_LCLRECIEVE – Indicates a receive capability for the local endpoint.
- IP_CAP_DIR_LCLRXTX – Indicates a receive and transmit capability for the local endpoint. Supported for T.38 only.

payload_type

The payload type. When using a standard payload type, set the value of this field to IP_USE_STANDARD_PAYLOADTYPE. When using a nonstandard payload type, use this field to specify the RTP payload type that will be used in conjunction with the coder specified in the capability field in this structure.

Not currently supported.

extra

The contents of the IP_CAPABILITY_UNION will be indicated by the type field.

rfu

Reserved for future use. Must be set to zero when not used.

IP_CAPABILITY_UNION

```
typedef union
{
    IP_AUDIO_CAPABILITY         audio;
    IP_VIDEO_CAPABILITY         video;
    IP_DATA_CAPABILITY          data;
} IP_CAPABILITY_UNION;
```

■ Description

The IP_CAPABILITY_UNION union enables different capability categories to define their own additional parameters or interest.

■ Field Descriptions

The fields of the IP_CAPABILITY_UNION union are described as follows:

audio
A structure that represents the audio capability. See IP_AUDIO_CAPABILITY, on page 383 for more information.

video
Not supported.

data
Not supported.

IP_CONNECT

```
typedef struct
{
   unsigned short     version;
   int                mediaHandle;
   int                faxHandle;
   eIPConnectType_e   connectType;
} IP_CONNECT;
```

Description

The IP_CONNECT data structure contains information required when associating a Media device with a T.38 Fax device required when switching from an audio coder to a T.38 coder and vice versa.

Field Descriptions

The fields of the IP_CONNECT data structure are described as follows:

version
: the current version number is 0x100

mediaHandle
: the Media device handle

faxHandle
: the T.38 Fax device handle

connectType
: the connection type. Possible values are:

- IP_FULLDUP
- IP_HALFDUP

Note: When disassociating a Media device from a T.38 Fax device, the faxHandle and connectType fields are ignored.

IP_DATA_CAPABILITY

```
typedef struct
{
   int      max_bit_rate;
} IP_DATA_CAPABILITY;
```

Description

The IP_DATA_CAPABILITY data structure provides additional information about the data capability.

Field Descriptions

The fields of the IP_DATA_CAPABILITY data structure are described as follows:

max_bit_rate

Possible values are:

- 2400
- 4800
- 9600
- 14400

The recommended value for T.38 coders is 14400.

IP_DTMF_DIGITS

```
typedef struct
{
    char          digit_buf[IP_MAX_DTMF_DIGITS];
    unsigned int  num_digits;
} IP_DTMF_DIGITS;
```

Description

The IP_DTMF_DIGITS data structure is used to provide DTMF information when the digits are received in a User Input Indication (UII) message with alphanumeric data.

Field Descriptions

The fields of the IP_DTMF_DIGITS data structure are described as follows:

digit_buf
The DTMF digit string buffer; 32 characters in size

num_digits
The number of DTMF digits in the string buffer

IP_H221NONSTANDARD

```
typedef struct
{
    int   country_code;
    int   extension;
    int   manufacturer_code;
} IP_H221NONSTANDARD;
```

Description

The IP_H221NONSTANDARD data structure is used to store H.221 nonstandard data.

Field Descriptions

The fields of the IP_H221NONSTANDARD data structure are described as follows:

country_code
The country code

extension
The extension number

manufacturer_code
The manufacturer code

IP_REGISTER_ADDRESS

```
typedef struct
{
    char              reg_client [IP_REG_CLIENT_ADDR_LENGTH];
    char              reg_server [IP_REG_SERVER_ADDR_LENGTH];
    int               time_to_live;
    int               max_hops;
} IP_REGISTER_ADDRESS;
```

■ Description

The IP_REGISTER_ADDRESS data structure is used to store registration information.

■ Field Descriptions

The fields of the IP_REGISTER_ADDRESS data structure are described as follows:

reg_client

The meaning is protocol dependent:

- When using H.323, this field is not used; any value specified is ignored
- When using SIP, this field is an alias for the subscriber

reg_server

The address of the registration server. Possible value are:

- An IP address in dot notation. A port number can also be specified as part of the address, for example, 10.242.212.216:1718.
- IP_REG_MULTICAST_DEFAULT_ADDR

time_to_live

The time to live value in seconds. The number of seconds for which a registration is considered to be valid when repetitive registration is selected.

In H.323, the default value of this field is 0, which disables repetitive registration.

In SIP, if this field is left at its default value 0, the call control library automatically enables auto-refresh with an Expires value of 3600 unless the application explicitly disables auto-refresh. Setting this to a non-zero value sets the Expires header in the REGISTER request to the specified value.

max_hops

The multicast time to live value in hops. The maximum number of hops (connections between routers) that a packet can take before being discarded or returned when using multicasting.

This field applies only to H.323 applications using gatekeeper discovery (H.225 RAS) via the default multicast registration address.

IP_TUNNELPROTOCOL_ALTID

```
typedef struct
{
    unsigned long  version;
    char           protocolType[MAX_TSM_ALTID_VARS_LENGTH];
    int            protocolTypeLength;
    char           protocolVariant[MAX_TSM_ALTID_VARS_LENGTH];
    int            protocolVariantLength;
    char           subIdentifier[MAX_TSM_ALTID_VARS_LENGTH];
    int            subIdentifierLength;
} IP_TUNNELPROTOCOL_ALTID;
```

■ Description

The IP_TUNNELPROTOCOL_ALTID data structure is used for H.323 Annex M tunneled signaling protocol alternate protocol ID information.

Applications should use the **INIT_IP_TUNNELPROTOCOL_ALTID( )** function to initialize the structure with the correct version number and initial field values.

■ Field Descriptions

The fields of the IP_TUNNELPROTOCOL_ALTID data structure are described as follows:

version
: the version number of the data structure. The correct value is set by the **INIT_IP_TUNNELPROTOCOL_ALTID( )** initialization function and should not be overridden.

protocolType
: a string that identifies the tunneled protocol type. Maximum length: 64

protocolTypeLength
: the length of the protocolType string

protocolVariant
: a string that identifies the tunneled protocol variant. Maximum length: 64

protocolVariantLength
: the length of the protocolVariant string

subIdentifier
: a string that provides additional tunneled protocol identification. Maximum length: 64

subIdentifierLength
: the length of the subIdentifier string

IP_VIRTBOARD

```
typedef struct
{
    unsigned short            version;
    unsigned int              total_max_calls;
    unsigned int              h323_max_calls;
    unsigned int              sip_max_calls;
    IP_ADDR                   localIP;
    unsigned short            h323_signaling_port;
    unsigned short            sip_signaling_port;
    void                      *reserved;
    unsigned short            size;
    unsigned int              sip_msginfo_mask;
    unsigned int              sup_serv_mask;
    unsigned int              h323_msginfo_mask;
    MIME_MEM                  sip_mime_mem
    unsigned short            terminal_type
    IP_ADDR                   outbound_proxy_IP
    unsigned short            outbound_proxy_port;
    char *                    outbound_proxy_hostname;
    EnumSIP_Enabled           E_SIP_tcpenabled;
    EnumSIP_TransportProtocol E_SIP_OutboundProxyTransport;
    EnumSIP_Persistence       E_SIP_Persistence;
    unsigned short            SIP_maxUDPmsgLen;
    EnumSIP_TransportProtocol E_SIP_DefaultTransport;
    EnumSIP_RequestRetry      E_SIP_RequestRetry;
    EnumSIP_Enabled           E_SIP_OPTIONS_Access;
    unsigned int              sip_registrar_registrations;
}IP_VIRTBOARD;
```

■ Description

The IP_VIRTBOARD data structure is used to store configuration and capability information about an IPT board device that is used when the device is started. An array of IP_VIRTBOARD structures (one for each virtual board in the system) is referenced by the IPCCLIB_START_DATA structure, which is passed to the **gc_Start( )** function. The IP_VIRTBOARD structure must be initialized to default values by the **INIT_IP_VIRTBOARD( )** initialization function; those default values can be overridden by the application before calling **gc_Start( )**.

■ Field Descriptions

The fields of the IP_VIRTBOARD data structure are described as follows:

version
: The version of the structure. The correct version number is populated by the **INIT_IP_VIRTBOARD( )** function and should not be overriden.

total_max_calls
: The maximum total number of IPT devices that can be open concurrently. Possible values are in the range 1 to 2016 (IP_CFG_MAX_AVAILABLE_CALLS). Each IPT device can support both the H.323 and SIP protocols. The default value is 120.

h323_max_calls
: The maximum number of IPT devices that can be used for H.323 calls. Possible values are in the range 1 to 2016 (IP_CFG_MAX_AVAILABLE_CALLS). The default value is 120.

sip_max_calls

The maximum number of IPT devices that can be used for SIP calls. Possible values are in the range 1 to 2016 (IP_CFG_MAX_AVAILABLE_CALLS). The default value is 120.

localIP

The local IP address of type IP_ADDR. See the reference page for IP_ADDR, on page 382.

h323_signaling_port

The H.323 call signaling port. Possible values are a valid port number or IP_CFG_DEFAULT. The default H.323 signaling port is 1720.

sip_signaling_port

The SIP call signaling port. Possible values are a valid port number or IP_CFG_DEFAULT. The default SIP signaling port is 5060.

reserved

For library use only

size

For library use only

sip_msginfo_mask (structure version ≥ 0x101 only)

Enables and disables access to SIP message information. Access is disabled by default. The following mask values, which may be OR'ed together, are defined to enable these features:

- IP_SIP_MSGINFO_ENABLE – enable access to supported SIP message information fields
- IP_SIP_MIME_ENABLE – enable sending and receiving of SIP messages that contain MIME information

sup_serv_mask (structure version ≥ 0x102 only)

Enables and disables the call transfer supplementary service. The service is disabled by default. Use the following value to enable the feature:

- IP_SUP_SERV_CALL_XFER – enable call transfer service

h323_msginfo_mask (structure version ≥ 0x103 only)

Enables and disables reception of H.323 message information. Access is disabled by default. Two mask values, which may be OR'ed together, are defined to enable the features:

- IP_H323_ANNEXMMSG_ENABLE – Enable reception of H.323 Annex M tunneled signaling messages in H.225 messages
- IP_H323_MSGINFO_ENABLE – enable access to H.323 message information fields

sip_mime_mem (structure version ≥ 0x104 only)

Sets the number and size of buffers that will be allocated for the MIME memory pool when the SIP MIME feature is enabled (no buffers are allocated if the feature is not enabled). The default values indicated below are set by the **INIT_MIME_MEM( )** macro, which is called by the **INIT_IP_VIRTBOARD( )** initialization function. The MIME_MEM data structure is defined as follows:

```
typedef struct
{
     unsigned short   version;     /* Version set by INIT_MIME_MEM   */
     unsigned int     size;        /* Default = 1500                 */
     unsigned int     number;      /* Default = (sip_max_calls * 5)  */
}MIME_MEM;
```

terminal_type (structure version ≥ 0x104 only)

Sets the Terminal Type for the virtual board which will be used during RAS registration (H.323 terminal type) and during Master Slave determination (H.245 terminal type). The value may only be changed from the default that is set by the **INIT_IP_VIRTBOARD( )** initialization function before calling **gc_Start( )**. Unsigned shorts from 0 to 255 are valid values, but the specific values 0 and 255 are reserved and will result in the terminal type being set to the default. Values larger than 255 will be truncated to 8 bits. The following symbolic values are defined:

- IP_TT_GATEWAY (Default) – Value = 60, for operation as terminal type Gateway
- IP_TT_TERMINAL – value = 50, for operation as terminal type Terminal

outbound_proxy_IP (structure version ≥ 0x105 only)

Sets the IP address of the SIP outbound proxy, which is used instead of the original Request URI for outbound SIP requests. The default value is 0, which disables outbound proxy unless the outbound_proxy_hostname field is set to a non-NULL name.

outbound_proxy_port (structure version ≥ 0x105 only)

Sets the port number of the SIP outbound proxy specified by outbound_proxy_IP. The default value is 5060, which is the same as the default SIP signaling port number.

outbound_proxy_hostname (structure version ≥ 0x105 only)

Sets the specified hostname as the SIP outbound proxy instead of a hard-coded IP address. If outbound_proxy_IP is set to 0, this hostname is resolved as the outbound proxy address. If outbound_proxy_IP is set to an IP address, this field is ignored and outbound_proxy_IP and outbound_proxy_port are used instead. The default value is NULL.

E_SIP_tcpenabled (structure version ≥ 0x106 only)

Enables the handling of incoming SIP messages that use TCP (received on the port number specified in sip_signaling_port), and the ability to specify TCP transport for SIP requests. The following symbolic values are defined:

- ENUM_Disabled (default) – disable TCP transport support (use default UDP transport)
- ENUM_Enabled – enable TCP transport support for incoming and outgoing messages

E_SIP_OutboundProxyTransport (structure version ≥ 0x106 only)

Selects the default transport protocol for SIP requests when an outbound proxy has been set up via the outbound_proxy_IP or outbound_proxy_hostname field (assuming that TCP is enabled via E_SIP_tcpenabled). The following symbolic values are defined:

- ENUM_TCP – use TCP protocol for the outbound proxy; if this value is set when TCP is not enabled or when TCP is enabled but no SIP proxy is configured, **gc_Start( )** returns an IPERR_BAD_PARM error
- ENUM_UDP (default) – use UDP protocol for the outbound proxy

E_SIP_Persistence (structure version ≥ 0x106 only)

Sets the persistence of TCP connections (assuming that TCP has been enabled via E_SIP_tcpenabled). This field has no effect on whether TCP is used for requests; it only affects the connections that are made when TCP is actually used. The following symbolic values are defined:

- ENUM_PERSISTENCE_NONE – no persistence; TCP connection is closed after each request
- ENUM_PERSISTENCE_TRANSACT – transaction persistence; TCP connection is closed after each transaction

- ENUM_PERSISTENCE_TRANSACT_USER (default) – user persistence; TCP connection is maintained for the lifetime of the “user” of the transaction (the CallLeg, for example)

SIP_maxUDPmsgLen (structure version ≥ 0x106 only)
Sets the maximum size for UDP SIP requests; above this threshold, the TCP transport protocol is automatically used instead of UDP (assuming that TCP is enabled via E_SIP_tcpenabled). The default value is 1300 (as recommended by RFC3261). Value may be set to 0 or VIRTBOARD_SIP_NOUDPMSGSIZECHECK to disable the size checking and reduce the message processing overhead.

E_SIP_DefaultTransport (structure version ≥ 0x106 only)
Sets the default transport protocol that is used when there is no proxy set (assuming that TCP is enabled by E_SIP_tcpenabled). The application can override the default for a particular request by explicitly specifying the transport protocol with a “transport= ” header parameter. The following symbolic values are defined:

- ENUM_TCP – use TCP unless “;transport=udp” is set by application; if this value is set when TCP is not enabled, **gc_Start( )** returns an IPERR_BAD_PARM error
- ENUM_UDP (default) – use UDP unless “;transport=tcp” is set by application

E_SIP_RequestRetry (structure version ≥ 0x107 only)
Sets the behavior that the SIP stack follows when a particular address-transport combination has failed for a SIP request; this may be a UDP failure after multiple retries or a TCP failure. The following symbolic values are defined:

- ENUM_REQUEST_RETRY_ALL (default) – there will be a retry if the DNS server has provided a list of IP addresses with transports, and there will also be a retry on the last (or only) address if the transport was TCP and the failure reason qualifies for retry
- ENUM_REQUEST_RETRY_DNS – there will be a retry if the DNS server has provided a list of IP addresses with transports
- ENUM_REQUEST_RETRY_FORCEDTCP – there will be a retry if the DNS server has provided a list of IP addresses with transports, and there will also be a retry on the last (or only) address if the transport was forced to be TCP because of message length and the failure reason qualifies for retry
- ENUM_REQUEST_RETRY_NONE – there will be no retry on request failure

E_SIP_OPTIONS_Access (structure version ≥ 0x108 only)
Enables application access to incoming OPTIONS, and the ability to send OPTIONS requests. The following symbolic values are defined:

- ENUM_Disabled (default) – disable application access to OPTIONS messages
- ENUM_Enabled – enable application access to OPTIONS messages

sip_registrar_registrations (structure version ≥ 0x109 only)
Specifies the number of unique SIP registrations that can be created. A unique registration is defined as a unique Address Of Record/Registrar pair, so registering the same AOR on a different Registrar is counted as a second unique registration. The range for this field is 1 to 10000. The default value is sip_max_calls (120).

IPCCLIB_START_DATA

```
typedef struct
{
    unsigned short    version;
    unsigned char     delimiter;
    unsigned char     num_boards;
    IP_VIRTBOARD      *board_list;
    unsigned long     max_parm_data_size;
} IPCCLIB_START_DATA;
```

■ Description

The IPCCLIB_START_DATA structure is used to configure the IP call control library when starting Global Call. The IPCCLIB_START_DATA structure is passed to the **cg_Start( )** function via the CCLIB_START_STRUCT and GC_START_STRUCT data structures. Applications must use the **INIT_IPCCLIB_START_DATA( )** function to populate a IPCCLIB_START_DATA structure with default values before overriding the default values as desired.

■ Field Descriptions

The fields of the IPCCLIB_START_DATA data structure are described as follows:

version
: The version of the start structure. The correct version number is populated by the **INIT_IPCCLIB_START_DATA( )** function and does not need to be overriden.

delimiter
: An ANSI character that specifies the address string delimiter; the default delimiter is the comma (,). The specified delimiter character is used to separate the components of the destination information when using **gc_MakeCall( )**, for example.

num_boards
: The number of IPT board devices. See Section 2.3.2, “IPT Board Devices”, on page 43 for more information on IPT board devices. The maximum value is 8, and the default is 2.

board_list
: A pointer to an array of IP_VIRTBOARD structures, one structure for each IPT board device. See IP_VIRTBOARD, on page 394 for more information.

max_parm_data_size (structure version ≥ 0x200)
: The maximum data size (in bytes) for Global Call parameters that support data lengths greater than 255 bytes. The default value for this field is 255 for backwards compatibility; the maximum value is 4096.

Note: The only Global Call parameter that currently supports >255 byte data is IPSET_SIP_MSGINFO / IPPARM_SIP_HDR.

RTP_ADDR

```
typedef struct
{
     int                version
     unsigned short     port;
     unsigned char      ip_ver;
     union
     {
          unsigned int      ipv4;
          unsigned int      ipv6[4];
     } u_ipaddr;
} RTP_ADDR, *RTP_ADDRP;
```

■ Description

The RTP_ADDR data structure contains a complete RTP address, which includes both the port number and the IP address. The RTP_ADDR structure is used when retrieving the local and remote RTP addresses from the Global Call completion event when a call is connected.

■ Field Descriptions

The fields of the RTP_ADDR data structure are described as follows:

version
: data structure version identification, for library use only

port
: the port number used by an RTP stream

ip_ver
: format of the IP address; currently, the only valid value is IPVER4

ipv4
: the IP address used by an RTP stream, in IPv4 format

ipv6[4]
: reserved for future use

IP-Specific Event Cause Codes 10

This chapter lists the IP-specific error and event cause codes and provides a description of each code. The codes described in this chapter are defined in the *gcip_defs.h* header file.

When a GCEV_DISCONNECTED event is received, use the **gc_ResultInfo( )** function to retrieve the reason or cause of that event.

When using **gc_DropCall( )** with H.323, only event cause codes prefixed by IPEC_H2250 or IPEC_Q931 should be specified in the **cause** parameter.

When using **gc_DropCall( )** with SIP, if the application wants to reject a call during call establishment, the relevant cause value for the **gc_DropCall( )** function can be either one of the generic Global Call cause values for dropping a call (see the **gc_DropCall( )** function description in the *Global Call API Library Reference*), or one of the cause codes prefixed by IPEC_SIP in this chapter. If the application wants to drop a call that is already connected (simply hanging up normally) the same rules apply, but the cause is not relevant in the BYE message.

When using **gc_Extension( )** to reject an incoming request to switch from audio to T.38 fax or vice versa, use only the cause codes prefixed by “IPEC_Q931Cause” for H.323, or the cause codes prefixed by “IPEC_SIPReason” for SIP.

10.1 IP-Specific Error Codes

The following IP-specific error codes are supported:

IPERR_ADDRESS_IN_USE
: The address specified is already in use. For IP networks, this will usually occur if an attempt is made to open a socket with a port that is already in use.

IPERR_ADDRESS_RESOLUTION
: Unable to resolve address to a valid IP address.

IPERR_BAD_PARAM
: Call failed because of a bad parameter.

IPERR_CALLER_ID
: Unable to allocate or copy caller ID string.

IPERR_CANT_CLOSE_CHANNEL
: As a result of the circumstances under which this channel was opened, it cannot be closed. This could occur for some protocols in the scenario when channels are opened before the call is connected. In this case, the channels should be closed and deleted after hang-up.

IPERR_CHANNEL_ACTIVE
: Media channel is already active.

IPERR_COPYING_OCTET_STRING
: Unable to copy octet string.

IPERR_COPYING_OR_RESOLVING_ALIAS
An error occurred while copying the alias. The error could be the result of a memory allocation failure or it could be an invalid alias format.

IPERR_DESTINATION_UNKNOWN
Failure to locate the host with the address given.

IPERR_DIAL_ADDR_MUST_BE_ALIAS
The address being dialed in this case may not be an IP address or domain name. It must be an alias because two intermediate addresses have already been specified, that is, Local Proxy, Remote Proxy and Gateway Address.

IPERR_DLL_LOAD_FAILED
Dynamic load of a DLL failed.

IPERR_DTMF_PENDING
Already in a DTMF generate state.

IPERR_DUP_CONF_ID
A conference ID was specified that matches an existing conference ID for another conference.

IPERR_FRAMESPERPACKET_NOT_SUPP
Setting frames-per-packet is not supported on the specified audio capability.

IPERR_GC_INVLINEDEV
Invalid line device.

IPERR_HOST_NOT_FOUND
Could not reach the party with the given host address.

IPERR_INCOMING_CALL_HANDLE
The handle passed as the incoming call handle does not refer to a valid incoming call.

IPERR_INTERNAL
An internal error occurred.

IPERR_INVALID_ADDRESS_TYPE
The address type specified did not map to any known address type.

IPERR_INVALID_CAPS
Channel open or response failed due to invalid capabilities.

IPERR_INVALID_DEST_ADDRESS
The destination address did not conform to the type specified.

IPERR_INVALID_DOMAIN_NAME
The domain name given is invalid.

IPERR_INVALID_DTMF_CHAR
Invalid DTMF character sent.

IPERR_INVALID_EMAIL_ADDRESS
The e-mail address given is invalid.

IPERR_INVALID_HOST_NAME
The host name given is invalid.

IPERR_INVALID_ID
An invalid ID was specified.

IPERR_INVALID_IP_ADDRESS
The IP address given is invalid.

IPERR_INVALID_MEDIA_HANDLE
The specified media handle is different from the already attached media handle.

IPERR_INVALID_PHONE_NUMBER
The phone number given is invalid.

IPERR_INVALID_PROPERTY
The property ID is invalid.

IPERR_INVALID_STATE
Invalid state to make this call.

IPERR_INVALID_URL_ADDRESS
The URL address given is invalid.

IPERR_INVDEVNAME
Invalid device name.

IPERR_IP_ADDRESS_NOT_AVAILABLE
The network socket layer reports that the IP address is not available. This can happen if the system does not have a correctly configured IP address.

IPERR_LOCAL_INTERNAL_PROXY_ADDR
Local internal proxy specified could not be resolved to a valid IP address or domain name.

IPERR_MEDIA_NOT_ATTACHED
No media resource was attached to the specified line device.

IPERR_MEMORY
Memory allocation failure.

IPERR_MULTIPLE_CAPS
Attaching a channel with multiple capabilities is not supported by this stack or it is not supported in this mode.

IPERR_MULTIPLE_DATATYPES
Attaching a channel with multiple data types (such as audio and video) is not permitted. All media types proposed for a single channel must be of the same type.

IPERR_NO_AVAILABLE_PROPOSALS
No available proposals to respond to.

IPERR_NO_CAPABILITIES_SPECIFIED
No capabilities have been specified yet. They must either be pre-configured in the configuration file or they must be set using an extended capability API.

IPERR_NO_DTMF_CAPABILITY
The remote endpoint does not have DTMF capability.

IPERR_NO_INTERSECTING_CAPABILITIES
No intersecting capability found.

IPERR_NOANSWER
Timeout due to no answer from peer.

IPERR_NOT_IMPLEMENTED
The function or property call has not been implemented. This differs from IPERR_UNSUPPORTED in that there is the implication that this is an early release which intends to implement the feature or function.

IPERR_NOT_MULTIPOINT_CAPABLE
The call cannot be accepted into a multipoint conference because there is no known multipoint controller, or the peer in a point-to-point conference is not multipoint capable.

IPERR_NULL_ADDRESS
Address given is NULL.

IPERR_NULL_ALIAS
The alias specified is NULL or empty.

IPERR_OK
Successful completion.

IPERR_PEER_REJECT
Peer has rejected the call placed from this endpoint.

IPERR_PENDING_RENEGOTIATION
A batched channel renegotiation is already pending. This implementation does not support queuing of batched renegotiation.

IPERR_PROXY_GATEWAY_ADDR
Two intermediate addresses were already specified in the local internal proxy and remote proxy addresses. The gateway address in this case cannot be used.

IPERR_REMOTE_PROXY_ADDR
Remote proxy specified could not be resolved to a valid IP address or domain name.

IPERR_SERVER_REGISTRATION_FAILED
Attempt to register with the registration and admission server (RAS) failed.

IPERR_STILL_REGISTERED
The address object being deleted is still registered and cannot be deleted until it is unregistered.

IPERR_TIMEOUT
Timeout occurred while executing an internal function.

IPERR_UNAVAILABLE
The requested data is unavailable.

IPERR_UNDELETED_OBJECTS
The object being deleted has child objects that have not been deleted.

IPERR_UNICODE_TO_ASCII
Unable to convert the string or character from unicode or wide character format to ASCII.

IPERR_UNINITIALIZED
The stack has not been initialized.

IPERR_UNKNOWN_API_GUID
This is the result of either passing in a bogus GUID or one that is not found in the current DLL or executable.

IPERR_UNRESOLVABLE_DEST_ADDRESS
No Gateway, Gatekeeper, or Proxy is specified, therefore the destination address must be a valid resolvable address. In the case of IP based call control, the address specified should be an IP address or a resolvable host or domain name.

IPERR_UNRESOLVABLE_HOST_NAME)
The host or domain name could not be resolved to a valid address. This will usually occur if the host or domain name is not valid or is not accessible over the existing network.

IPERR_UNSUPPORTED
This function or property call is unsupported in this configuration or implementation of stack. This differs from IPERR_NOT_IMPLEMENTED in that it implies no future plan to support this feature of property.

10.2 Error Codes When Using H.323

The following error codes are supported:

IPEC_addrRegistrationFailed
Registration with the Registration and Admission server failed.

IPEC_addrListenFailed
Stack was unable to register to listen for incoming calls.

IPEC_CHAN_REJECT_unspecified
No cause for rejection specified.

IPEC_CHAN_REJECT_dataTypeNotSupported
The terminal was not capable of supporting the dataType indicated in OpenLogicalChannel.

IPEC_CHAN_REJECT_dataTypeNotAvailable
The terminal was not capable of supporting the dataType indicated in OpenLogicalChannel simultaneously with the dataTypes of logical channels that are already open.

IPEC_CHAN_REJECT_unknownDataType
The terminal did not understand the dataType indicated in OpenLogicalChannel.

IPEC_CHAN_REJECT_insuffientBandwdith
The channel could not be opened because permission to use the requested bandwidth for the logical channel was denied.

IPEC_CHAN_REJECT_unsuitableReverseParameters
This code shall only be used to reject a bi-directional logical channel request when the only reason for rejection is that the requested parameters are inappropriate.

IPEC_CHAN_REJECT_dataTypeALCombinationNotSupported
The terminal was not capable of supporting the dataType indicated in OpenLogicalChannel simultaneously with the Adaptation Layer type indicated in H223LogicalChannelParameters.

IPEC_CHAN_REJECT_multicastChannelNotAllowed
Multicast Channel could not be opened.

IPEC_CHAN_REJECT_separateStackEstablishmentFailed
A request to run the data portion of a call on a separate stack failed.

IPEC_CHAN_REJECT_invalidSessionID
Attempt by the slave to set the SessionID when opening a logical channel to the master.

IPEC_CHAN_REJECT_masterSlaveConflict
Attempt by the slave to open logical channel in which the master has determined a conflict may occur.

IPEC_CHAN_REJECT_waitForCommunicationMode
Attempt to open a logical channel before the MC has transmitted the CommunicationModeCommand.

IPEC_CHAN_REJECT_invalidDependentChannel
Attempt to open a logical channel with a dependent channel specified that is not present.

IPEC_CHAN_REJECT_replacementForRejected
A logical channel of the type attempted cannot be opened using the replacement **For** parameter.The transmitter may wish to re-try by first closing the logical channel that is to be replaced, and then opening the replacement.

IPEC_CALL_END_timeout
A callback was received because a local timer expired.

IPEC_InternalError
An internal error occurred while executing asynchronously.

IPEC_INFO_NONE_NOMORE
No more digits are available.

IPEC_INFO_PRESENT_MORE
The requested digits are now available. More/additional digits are available.

IPEC_INFO_PRESENT_ALL
The requested digits are now available.

IPEC_INFO_NONE_TIMEOUT
No digits are available; timed out.

IPEC_INFO_SOME_NOMORE
Only some digits are available, no more digits will be received.

IPEC_INFO_SOME_TIMEOUT
Only some digits are available; timed out.

IPEC_NO_MATCHING_CAPABILITIES
No intersection was found between the proposed and matching capabilities.

IPEC_REG_FAIL_duplicateAlias
The alias used to register with the Registration and Admission server is already registered. This failure typically results if the endpoint is already registered. It could also occur with some servers if a registration is attempted too soon after unregistering using the same alias.

IPEC_REG_FAIL_invalidCallSigAddress
Server registration failed due to an invalid call signalling address specified.

IPEC_REG_FAIL_invalidAddress
The local host address specified for communicating with the server is invalid.

IPEC_REG_FAIL_invalidAlias
The alias specified did not conform to the format rules for the type of alias specified.

IPEC_REG_FAIL_invalidTermType
An invalid terminal type was specified with the registration request.

IPEC_REG_FAIL_invalidTransport
The transport type of the local host's address is not supported by the server.

IPEC_REG_FAIL_qosNotSupported
The registration request announced a transport QoS that was not supported by the server.

IPEC_REG_FAIL_reRegistrationRequired
Registration permission has expired. Registration should be performed again.

IPEC_REG_FAIL_resourcesUnavailable
The server rejected the registration request due to unavailability of resources. This typically occurs if the server has already reached the maximum number of registrations it was configured to accept.

IPEC_REG_FAIL_securityDenied
The server denied access for security reasons. This can occur if the password supplied does not match the password on file for the alias being registered.

IPEC_REG_FAIL_unknown
The server refused to allow registration for an unknown reason.

IPEC_REG_FAIL_serverDown
The server has gone down or is no longer responding.

IPEC_MEDIA_startSessionFailed
Attempt to call **gc_media_StartSession( )** (an internal function) after establishing media channel returned error.

IPEC_MEDIA_TxFailed
Attempt to establish or terminate a Tx channel with attached capabilities failed. The application is expected to keep the Rx capabilities unchanged in the next call to **gc_AttachEx**( ).

IPEC_MEDIA_RxFailed
Attempt to establish or terminate an Rx channel with attached capabilities failed. The application is expected to keep the Tx capabilities unchanged in the next call to **gc_AttachEx**( ).

IPEC_MEDIA_TxRxFailed
Attempts to establish or terminate Tx and Rx channels with attached capabilities failed.

IPEC_MEDIA_OnlyTxFailed
Attempts to establish a Tx channel with attached capabilities failed. The status of other media channel is unavailable. Relevant to the GCEV_MEDIA_REJ event.

IPEC_MEDIA_OnlyRxFailed
Attempts to establish an Rx channel with attached capabilities failed. The status of other media channel is unavailable. Relevant to the GCEV_MEDIA_REJ event.

IPEC_MEDIA_TxRequired
Attempts to establish a Tx channel with attached capabilities failed.

IPEC_MEDIA_RxRequired
Attempts to establish an Rx channel with attached capabilities failed.

IPEC_TxRx_Fail
Both channels have failed to open.

IPEC_Tx_FailTimeout
A Tx channel failed to open because of timeout.

IPEC_Rx_FailTimeout
An Rx channel failed to open because of timeout.

IPEC_Tx_Fail
A Tx channel failed to open for an unknown reason.

IPEC_Rx_Fail
An Rx channel failed to open for an unknown reason.

IPEC_TxRx_FailTimeout
Both the Tx and Rx channels failed because of a timeout.

IPEC_TxRx_Rej
Both the Tx and Rx channels were rejected for an unknown reason.

IPEC_Tx_Rej
Opening of a Tx channel was rejected for unknown reasons.

IPEC_Rx_Rej
Opening of an Rx channel was rejected for unknown reasons.

IPEC_CHAN_FAILURE_unspecified
The channel failed to open/close because of an unspecified reason.

IPEC_CHAN_FAILURE_timeout
The channel failed to open/close because of a timeout.

IPEC_CHAN_FAILURE_localResources
The channel failed to open/close because of limited resources.

IPEC_FAIL_TxRx_unspecified
Both the Tx and Rx channels failed to open for unspecified reasons.

IPEC_FAIL_TxUnspecifiedRxTimeout
A Tx channel failed to open for unspecified reasons and the Rx channel failed to open because of a timeout.

IPEC_FAILTxUnspecifiedRxResourceUnsuff
A Tx channel failed to open for unspecified reasons and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_RxUnspecifiedTxTimeout
An Rx channel failed to open for unspecified reasons and the Tx channel failed to open because of a timeout.

IPEC_FAIL_RXUnspecifiedTxResourceUnsuff
An Rx channel failed to open for unspecified reasons and the Tx channel failed to open because of insufficient resources.

IPEC_FAIL_TxTimeoutRxUnspecified
A Tx channel failed to open because of a timeout and the Rx channel failed to open for unspecified reasons.

IPEC_FAIL_TxRxTimeout
The Tx and Rx channels both failed to open because of a timeout.

IPEC_FAIL_TxTimeoutRxResourceUnsuff
A Tx channel failed to open because of a timeout and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_RxTimeoutTXUnspecified
An Rx channel failed because of a timeout and the Tx channel failed for unspecified reasons.

IPEC_FAIL_RxTimeoutTxResourceUnsuff
A Tx channel failed to open because of a timeout and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_TxResourceUnsuffRxUnspecified
A Tx channel failed to open because of insufficient resources and the Rx channel failed to open for unspecified reasons.

IPEC_FAIL_TxResourceUnsuffRxTimeout
A Tx channel failed to open because of insufficient resources and the Rx channel failed to open because of a timeout.

IPEC_FAIL_TxRxResourceUnsuff
Tx and Rx channels failed to open because of insufficient resources.

IPEC_FAIL_RxResourceUnsuffTxUnspecified
A Tx channel failed to open for unspecified reasons and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_RxResourceUnsuffTxTimeout
A Tx channel failed to open because of a timeout and the Rx channel failed to open because of insufficient resources.

10.3 Internal Disconnect Reasons

The following internal disconnect reasons are supported when using H.323:

IPEC_InternalReasonBusy (0x3e9, 1001 decimal)
Cause 01; Busy

IPEC_InternalReasonCallCompletion (0x3ea, 1002 decimal)
Cause 02; Call Completion

IPEC_InternalReasonCanceled (0x3eb, 1003 decimal)
Cause 03; Cancelled

IPEC_InternalReasonCongestion (0x3ec, 1004 decimal)
Cause 04; Network congestion

IPEC_InternalReasonDestBusy (0x3ed, 1005 decimal)
Cause 05; Destination busy

IPEC_InternalReasonDestAddrBad (0x3ee, 1006 decimal)
Cause 06; Invalid destination address

IPEC_InternalReasonDestOutOfOrder (0x3ef, 1007 decimal)
Cause 07; Destination out of order

IPEC_InternalReasonDestUnobtainable (0x3f0, 1008 decimal)
Cause 08; Destination unobtainable

IPEC_InternalReasonForward (0x3f1, 1009 decimal)
Cause 09; Forward

IPEC_InternalReasonIncompatible (0x3f2, 1010 decimal)
Cause 10; Incompatible

IPEC_InternalReasonIncomingCall, (0x3f3, 1011 decimal)
Cause 11; Incoming call

IPEC_InternalReasonNewCall (0x3f4, 1012 decimal)
Cause 12; New call

IPEC_InternalReasonNoAnswer (0x3f5, 1013 decimal)
Cause 13; No answer from user

IPEC_InternalReasonNormal (0x3f6, 1014 decimal)
Cause 14; Normal clearing

IPEC_InternalReasonNetworkAlarm (0x3f7, 1015 decimal)
Cause 15; Network alarm

IPEC_InternalReasonPickUp (0x3f8, 1016 decimal)
Cause 16; Pickup

IPEC_InternalReasonProtocolError (0x3f9, 1017 decimal)
Cause 17; Protocol error

IPEC_InternalReasonRedirection (0x3fa, 1018 decimal)
Cause 18; Redirection

IPEC_InternalReasonRemoteTermination (0x3fb, 1019 decimal)
Cause 19; Remote termination

IPEC_InternalReasonRejection (0x3fc, 1020 decimal)
Cause 20; Call rejected

IPEC_InternalReasonSIT (0x3fd, 1021 decimal)
Cause 21; Special Information Tone (SIT)

IPEC_InternalReasonSITCustIrreg (0x3fe, 1022 decimal)
Cause 22; SIT, Custom Irregular

IPEC_InternalReasonSITNoCircuit (0x3ff, 1023 decimal)
Cause 23; SIT, No Circuit

IPEC_InternalReasonSITReorder (0x400, 1024 decimal)
Cause 24; SIT, Reorder

IPEC_InternalReasonTransfer (0x401, 1025 decimal)
Cause 25; Transfer

IPEC_InternalReasonUnavailable (0x402, 1026 decimal)
Cause 26; Unavailable

IPEC_InternalReasonUnknown (0x403, 1027 decimal)
Cause 27; Unknown cause

IPEC_InternalReasonUnallocatedNumber (0x404, 1028 decimal)
Cause 28; Unallocated number

IPEC_InternalReasonNoRoute (0x405, 1029 decimal)
Cause 29; No route

IPEC_InternalReasonNumberChanged (0x406, 1030 decimal)
Cause 30; Number changed

IPEC_InternalReasonOutOfOrder (0x407, 1031 decimal)
Cause 31; Destination out of order

IPEC_InternalReasonInvalidFormat (0x408, 1032 decimal)
Cause 32; Invalid format

IPEC_InternalReasonChanUnavailable (0x409, 1033 decimal)
Cause 33; Channel unavailable

IPEC_InternalReasonChanUnacceptable (0x40a, 1034 decimal)
Cause 34; Channel unacceptable

IPEC_InternalReasonChanNotImplemented (0x40b, 1035 decimal)
Cause 35; Channel not implemented

IPEC_InternalReasonNoChan (0x40c, 1036 decimal)
Cause 36; No channel

IPEC_InternalReasonNoResponse (0x40d, 1037 decimal)
Cause 37; No response

IPEC_InternalReasonFacilityNotSubscribed (0x40e, 1038 decimal)
Cause 38; Facility not subscribed

IPEC_InternalReasonFacilityNotImplemented (0x40f, 1039 decimal)
Cause 39; Facility not implemented

IPEC_InternalReasonServiceNotImplemented (0x410, 1040 decimal)
Cause 40; Service not implemented

IPEC_InternalReasonBarredInbound (0x411, 1041 decimal)
Cause 41; Barred inbound calls

IPEC_InternalReasonBarredOutbound (0x412, 1042 decimal)
Cause 42; Barred outbound calls

IPEC_InternalReasonDestIncompatible (0x413, 1043 decimal)
Cause 43; Destination incompatible

IPEC_InternalReasonBearerCapUnavailable (0x414, 1044 decimal)
Cause 44; Bearer capability unavailable

10.4 Event Cause Codes and Failure Reasons When Using H.323

The following event cause codes apply when using H.323.

H.225.0 Cause Codes

IPEC_H2250ReasonNoBandwidth (0x7d0, 2000 decimal)
Maps to Q.931/Q.850 cause 34 - No circuit or channel available; indicates that there is no appropriate circuit/channel presently available to handle the call.

IPEC_H2250ReasonGatekeeperResource (0x7d1, 2001 decimal)
Maps to Q.931/Q.850 cause 47 - Resource unavailable; used to report a resource unavailable event only when no other cause in the resource unavailable class applies.

IPEC_H2250ReasonUnreachableDestination (0x7d2, 2002 decimal)
Maps to Q.931/Q.850 cause 3 - No route to destination; indicates that the called party cannot be reached because the network through which the call has been routed does not serve the destination desired.

IPEC_H2250ReasonDestinationRejection (0x7d3, 2003 decimal)
Maps to Q.931/Q.850 cause 16 - Normal call clearing - indicates that the call is being cleared because one of the users involved in the call has requested that the call be cleared.

IPEC_H2250ReasonInvalidRevision (0x7d4, 2004 decimal)
Maps to Q.931/Q.850 cause 88 - Incompatible destination; indicates that the equipment sending this cause has received a request to establish a call which has low layer compatibility, high layer compatibility, or other compatibility attributes (for example, data rate) which cannot be accommodated.

IPEC_H2250ReasonNoPermission (0x7d5, 2005 decimal)
Maps to Q.931/Q.850 cause 111 - Interworking, unspecified.

IPEC_H2250ReasonUnreachableGatekeeper (0x7d6, 2006 decimal)
Maps to Q.931/Q.850 cause 38 - Network out of order; indicates that the network is not functioning correctly and that the condition is likely to last a relatively long period of time, for example, immediately re-attempting the call is not likely to be successful.

IPEC_H2250ReasonGatewayResource (0x7d7, 2007 decimal)
Maps to Q.931/Q.850 cause 42 - Switching equipment congestion; indicates that the switching equipment generating this cause is experiencing a period of high traffic.

IPEC_H2250ReasonBadFormatAddress (0x7d8, 2008 decimal)
Maps to Q.931/Q.850 cause 28 - Invalid number format; indicates that the called party cannot be reached because the called party number is not in a valid format or is incomplete.

IPEC_H2250ReasonAdaptiveBusy (0x7d9, 2009 decimal)
Maps to Q.931/Q.850 cause 41 - Temporary failure; indicates that the network is not functioning correctly and that the condition is not likely to last for a long period of time, for example, the user may wish to try another call attempt almost immediately.

IPEC_H2250ReasonInConf (0x7da, 2010 decimal)
Maps to Q.931/Q.850 cause 17 - User busy; used to indicate that the called party is unable to accept another call because the user busy condition has been encountered. This cause value may be generated by the called user or by the network.

IPEC_H2250ReasonUndefinedReason (0x7db, 2011 decimal)
Maps to Q.931/Q.850 cause 31 - Normal, unspecified; Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

IPEC_H2250ReasonFacilityCallDeflection (0x7dc, 2012 decimal)
Maps to Q.931/Q.850 cause 16 - Normal call clearing - indicates that the call is being cleared because one of the users involved in the call has requested that the call be cleared.

IPEC_H2250ReasonSecurityDenied (0x7dd, 2013 decimal)
Maps to Q.931/Q.850 cause 31 - Normal, unspecified; Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

IPEC_H2250ReasonCalledPartyNotRegistered (0x7de, 2014 decimal)
Maps to Q.931/Q.850 cause 20 - Subscriber absent; used when a mobile station has logged off, radio contact is not obtained with a mobile station or if a personal telecommunication user is temporarily not addressable at any user-network interface.

IPEC_H2250ReasonCallerNotRegistered (0x7df, 2015 decimal)
Maps to Q.931/Q.850 cause 31 - Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

Q.931 Cause Codes

IPEC_Q931Cause01UnassignedNumber (0xbb9, 3001 decimal)
Q.931 cause 01 - Unallocated (unassigned) number; indicates that the called party cannot be reached because. Although the called party number is in a valid format, it is not currently allocated (assigned).

IPEC_Q931Cause02NoRouteToSpecifiedTransitNetwork (0xbba, 3002 decimal)
Q.931 cause 02 - No route to specified transit network (national use); indicates that the equipment sending this cause has received a request to route the call through a particular transit network which it does not recognize. The equipment sending this cause does not recognize the transit network either because the transit network does not exist or because that particular transit network, while it does exist, does not serve the equipment which is sending this cause. This cause is supported on a network-dependent basis.

IPEC_Q931Cause03NoRouteToDestination (0xbbb, 3003 decimal)
Q.931 cause 03 - No route to destination; indicates that the called party cannot be reached because the network through which the call has been routed does not serve the destination desired. This cause is supported on a network-dependent basis.

IPEC_Q931Cause06ChannelUnacceptable (0xbbe, 3006 decimal)
Q.931 cause 06 - Channel unacceptable; indicates that the channel most recently identified is not acceptable to the sending entity for use in this call.

IPEC_Q931Cause07CallAwardedAndBeingDeliveredInAnEstablishedChannel (0xbbf, 3007 decimal)

Q.931 cause 07 - Call awarded and being delivered in an established channel; indicates that the user has been awarded the incoming call, and that the incoming call is being connected to a channel already established to that user for similar calls (e.g. packet-mode X.25 virtual calls).

IPEC_Q931Cause16NormalCallClearing (0xbc8, 3016 decimal)

Q.931 cause 16 - Normal call clearing; indicates that the call is being cleared because one of the user's involved in the call has requested that the call be cleared. Under normal situations, the source of this cause is not the network.

IPEC_Q931Cause17UserBusy (0xbc9, 3017 decimal)

Q.931 cause 17 - User busy; used to indicate that the called party is unable to accept another call because the user busy condition has been encountered. This cause value may be generated by the called user or by the network.

IPEC_Q931Cause18NoUserResponding (0xbca, 3018 decimal)

Q.931 cause 18 - No user responding; used when a called party does not respond to a call establishment message with either an alerting or connect indication within the prescribed period of time allocated.

IPEC_Q931Cause19UserAlertingNoAnswer (0xbcb, 3019 decimal)

Q.931 cause 19 - No answer from user (user alerted); used when the called party has been alerted but does not respond with a connect indication within a prescribed period of time. This cause is not necessarily generated by Q.931 procedures but may be generated by internal network timers.

IPEC_Q931Cause21CallRejected (0xbcd, 3021 decimal)

Q.931 cause 21 - Call rejected; indicates that the equipment sending this cause does not wish to accept this call, although it could have accepted the call because the equipment sending this cause is neither busy nor incompatible. This cause may also be generated by the network, indicating that the call was cleared due to a supplementary service constraint. The diagnostic field may contain additional information about the supplementary service and reason for rejection.

IPEC_Q931Cause22NumberChanged (0xbce, 3022 decimal)

Q.931 cause 22 - Number changed; returned to a calling party when the called party number indicated by the calling party is no longer assigned. The new called party number may optionally be included in the diagnostic field. If a network does not support this cause value, cause No. 1, unallocated (unassigned) number should be used.

IPEC_Q931Cause26NonSelectUserClearing (0xbd2, 3026 decimal)

Q.931 cause 26 - Non-selected user clearing; indicates that the user has not been awarded the incoming call.

IPEC_Q931Cause27DestinationOutOfOrder (0xbd3, 3027 decimal)

Q.931 cause 27 - Destination out of order; indicates that the destination indicated by the user cannot be reached because the interface to the destination is not functioning correctly. The term "not functioning correctly" indicates that a signalling message was unable to be delivered to the remote party, for example, a physical layer or data link layer failure at the remote party, or user equipment off-line.

IPEC_Q931Cause28InvalidNumberFormatIncompleteNumber (0xbd4, 3028 decimal)

Q.931 cause 28 - Invalid number format (address incomplete); indicates that the called party cannot be reached because the called party number is not in a valid format or is not complete. Note: This condition may be determined immediately after reception of an ST signal or on time-out after the last received digit.

IPEC_Q931Cause29FacilityRejected (0xbd5, 3029 decimal)

Q.931 cause 29 - Facility rejected; returned when a supplementary service requested by the user cannot be provided by the network.

IPEC_Q931Cause30ResponseToSTATUSENQUIRY (0xbd6, 3030 decimal)

Q.931 cause 30 - Response to STATUS ENQUIRY; included in the STATUS message when the reason for generating the STATUS message was the prior receipt of a STATUS ENQUIRY message.

IPEC_Q931Cause31NormalUnspecified (0xbd7, 3031 decimal)

Q.931 cause 31 - Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

IPEC_Q931Cause34NoCircuitChannelAvailable (0xbda, 3034 decimal)

Q.931 cause 34 - No circuit/channel available; indicates that there is no appropriate circuit/channel presently available to handle the call.

IPEC_Q931Cause38NetworkOutOfOrder (0xbde, 3038 decimal)

Q.931 cause 38 - Network out of order; indicates that the network is not functioning correctly and that the condition is likely to last a relatively long period of time, that is, immediately re-attempting the call is not likely to be successful.

IPEC_Q931Cause41TemporaryFailure (0xbe1, 3041 decimal)

Q.931 cause 41 - Temporary failure; indicates that the network is not functioning correctly and that the condition is not likely to last a long period of time, that is, the user may wish to try another call attempt almost immediately.

IPEC_Q931Cause42SwitchingEquipmentCongestion (0xbe2, 3042 decimal)

Q.931 cause 42 - Switching equipment congestion; indicates that the switching equipment generating this cause is experiencing a period of high traffic.

IPEC_Q931Cause43AccessInformationDiscarded (0xbe3, 3043 decimal)

Q.931 cause 43 - Access information discarded; indicates that the network could not deliver access information to the remote user as requested, that is, user-to-user information, low layer compatibility, high layer compatibility, or sub-address, as indicated in the diagnostic. The particular type of access information discarded is optionally included in the diagnostic.

IPEC_Q931Cause44RequestedCircuitChannelNotAvailable (0xbe4, 3044 decimal)

Q.931 cause 44 - Requested circuit/channel not available; returned when the circuit or channel indicated by the requesting entity cannot be provided by the other side of the interface.

IPEC_Q931Cause47ResourceUnavailableUnspecified (0xbe7, 3047 decimal)

Q.931 cause 47 - Resource unavailable, unspecified; used to report a resource unavailable event only when no other cause in the resource unavailable class applies.

IPEC_Q931Cause57BearerCapabilityNotAuthorized (0xbf1, 3057 decimal)

Q.931 cause 57 - Bearer capability not authorized; indicates that the user has requested a bearer capability that is implemented by the equipment that generated this cause but the user is not authorized to use.

IPEC_Q931Cause58BearerCapabilityNotPresentlyAvailable (0xbf2, 3058 decimal)

Q.931 cause 58 - Bearer capability not presently available; indicates that the user has requested a bearer capability that is implemented by the equipment that generated this cause but it is not available at this time.

IPEC_Q931Cause63ServiceOrOptionNotAvailableUnspecified (0xbf7, 3063 decimal)

Q.931 cause 63 - Service or option not available, unspecified; used to report a service or option not available event only when no other cause in the service or option not available class applies.

IPEC_Q931Cause65BearCapabilityNotImplemented (0xbf9, 3065 decimal)

Q.931 cause 65 - Bearer capability not implemented; indicates that the equipment sending this cause does not support the bearer capability requested.

IPEC_Q931Cause66ChannelTypeNotImplemented (0xbfa, 3066 decimal)

Q.931 cause 66 - Channel type not implemented; indicates that the equipment sending this cause does not support the channel type requested.

IPEC_Q931Cause69RequestedFacilityNotImplemented (0xbfd, 3069 decimal)

Q.931 cause 69 - Requested facility not implemented; indicates that the equipment sending this cause does not support the requested supplementary service.

IPEC_Q931Cause70OnlyRestrictedDigitalInformationBearerCapabilityIsAvailable (0xbfe, 3070 decimal)

Q.931 cause 70 - Only restricted digital information bearer capability is available (national use); indicates that the calling party has requested an unrestricted bearer service but that the equipment sending this cause only supports the restricted version of the requested bearer capability.

IPEC_Q931Cause79ServiceOrOptionNotImplementedUnspecified (0xc07, 3079 decimal)

Q.931 cause 79 - Service or option not implemented, unspecified; used to report a service or option not implemented event only when no other cause in the service or option not implemented class applies.

IPEC_Q931Cause81InvalidCallReferenceValue (0xc09, 3081 decimal)

Q.931 cause 81 - Invalid call reference value; indicates that the equipment sending this cause has received a message with a call reference that is not currently in use on the user-network interface.

IPEC_Q931Cause82IdentifiedChannelDoesNotExist (0xc0a, 3082 decimal)

Q.931 cause 82 - Identified channel does not exist; indicates that the equipment sending this cause has received a request to use a channel not activated on the interface for a call. For example, if a user has subscribed to those channels on a primary rate interface numbered from 1 to 12 and the user equipment or the network attempts to use channels 13 through 23, this cause is generated.

IPEC_Q931Cause83AsuspendedCallExistsButThisCallIdentityDoesNot (0xc0b, 3083 decimal)

Q.931 cause 83 - A suspended call exists, but this call identity does not; indicates that a call resume has been attempted with a call identity that differs from that in use for any presently suspended call(s).

IPEC_Q931Cause84CallIdentityInUse (0xc0c, 3084 decimal)

Q.931 cause 84 - Call identity in use; indicates that the network has received a call suspended request containing a call identity (including the null call identity) that is already in use for a suspended call within the domain of interfaces over which the call might be resumed.

IPEC_Q931Cause85NoCallSuspended (0xc0d, 3085 decimal)
Q.931 cause 85 - No call suspended; indicates that the network has received a call resume request containing a call identity information element that presently does not indicate any suspended call within the domain of interfaces over which calls may be resumed.

IPEC_Q931Cause86CallHavingTheRequestedCallIdentityHasBeenCleared (0xc0e, 3086 decimal)
Q.931 cause 86 - Call having the requested call identity has been cleared; indicates that the network has received a call resume request containing a call identity information element indicating a suspended call that has in the meantime been cleared while suspended (either by network timeout or by the remote user).

IPEC_Q931Cause88IncompatibleDestination (0xc10, 3088 decimal)
Q.931 cause 88 - Incompatible destination; indicates that the equipment sending this cause has received a request to establish a call that has low layer compatibility, high layer compatibility, or other compatibility attributes (for example, data rate) that cannot be accommodated.

IPEC_Q931Cause91InvalidTransitNetworkSelection (0xc13, 3091 decimal)
Q.931 cause 91 - Invalid transit network selection (national use); indicates that a transit network identification was received that is of an incorrect format as defined by Annex C/Q.931.

IPEC_Q931Cause95InvalidMessageUnspecified (0xc17, 3095 decimal)
Q.931 cause 95 - Invalid message, unspecified; used to report an invalid message event only when no other cause in the invalid message class applies.

IPEC_Q931Cause96MandatoryInformationElementMissing (0xc18, 3096 decimal)
Q.931 cause 96 - Mandatory information element is missing; indicates that the equipment sending this cause has received a message that is missing an information element that must be present in the message before that message can be processed.

IPEC_Q931Cause97MessageTypeNonExistentOrNotImplemented (0xc19, 3097 decimal)
Q.931 cause 97 - Message type non-existent or not implemented; indicates that the equipment sending this cause has received a message with a message type it does not recognize either because 1) the message type is not defined or 2) the message type is defined but not implemented by the equipment sending this cause.

IPEC_Q931Cause100InvalidInformationElementContents (0xc1c, 3100 decimal)
Q.931 cause 100 - Invalid information element contents; indicates that the equipment sending this cause has received an information element that it has implemented; however, one or more fields in the information element are coded in such a way that has not been implemented by the equipment sending this cause.

IPEC_Q931Cause101MessageNotCompatibleWithCallState (0xc1d, 3101 decimal)
Q.931 cause 101 - Message not compatible with call state; indicates that a message that is incompatible with the call state has been received.

IPEC_Q931Cause102RecoveryOnTimeExpiry (0xc1e, 3102 decimal)
Q.931 cause 102 - Recovery on timer expiry; indicates that a procedure has been initiated by the expiry of a timer in association with error handling procedures.

IPEC_Q931Cause111ProtocolErrorUnspecified (0xc27, 3111 decimal)
Q.931 cause 111 - Protocol error, unspecified; used to report a protocol error event only when no other cause in the protocol error class applies.

IPEC_Q931Cause127InterworkingUnspecified (0xc37, 3127 decimal)
Q.931 cause 127 - Interworking, unspecified; indicates that there has been interworking with a network that does not provide causes for the actions it takes. Thus, the precise cause for a message that is being sent cannot be ascertained.

RAS Failure Reasons

IPEC_RASReasonResourceUnavailable (0xfa1, 4001 decimal)
Resources have been exhausted. (In GRJ, RRJ, ARJ, and LRJ messages.)

IPEC_RASReasonInsufficientResources (0xfa2, 4002 decimal)
Insufficient resources to complete the transaction. (In BRJ messages.)

IPEC_RASReasonInvalidRevision (0xfa3, 4003 decimal)
The registration version is invalid. (In GRJ, RRJ, and BRJ messages.)

IPEC_RASReasonInvalidCallSignalAddress (0xa4, 4004 decimal)
The call signal address is invalid. (In RRJ messages.)

IPEC_RASReasonInvalidIPEC_RASAddress (0xfa5, 4005 decimal)
The supplied address is invalid. (In RRJ messages.)

IPEC_RASReasonInvalidTerminalType (0xfa6, 4006 decimal)
The terminal type is invalid. (In RRJ messages.)

IPEC_RASReasonInvalidPermission (0xfa7, 4007 decimal)
Permission has expired. (In ARJ messages.)

A true permission violation. (In BRJ messages.)

Exclusion by administrator or feature. (In LRJ messages.)

IPEC_RASReasonInvalidConferenceID (0xfa8, 4008 decimal)
Possible revision. (In BRJ messages.)

IPEC_RASReasonInvalidEndpointID (0xfa9, 4009 decimal)
The endpoint registration ID is invalid. (In ARJ messages.)

IPEC_RASReasonCallerNotRegistered (0xfaa, 4010 decimal)
The call originator is not registered. (In ARJ messages.)

IPEC_RASReasonCalledPartyNotRegistered (0xfab, 4011 decimal)
Unable to translate the address. (In ARJ messages.)

IPEC_RASReasonDiscoveryRequired (0xfac, 4012 decimal)
Registration permission has expired. (In RRJ messages.)

IPEC_RASReasonDuplicateAlias (0xfad, 4013 decimal)
The alias is registered to another endpoint. (In RRJ messages.)

IPEC_RASReasonTransportNotSupported (0xfae, 4014 decimal)
One or more of the transport addresses are not supported. (In RRJ messages.)

IPEC_RASReasonCallInProgress (0xfaf, 4015 decimal)
A call is already in progress. (In URJ messages.)

IPEC_RASReasonRouteCallToGatekeeper (0xfb0, 4016 decimal)
The call has been routed to a gatekeeper. (In ARJ messages.)

IPEC_RASReasonRequestToDropOther (0xfb1, 4017 decimal)
Unable to request to drop the call for others. (In DRJ messages.)

IPEC_RASReasonNotRegistered (0xfb2, 4018 decimal)
Not registered with a gatekeeper. (In DRJ, LRJ, and INAK messages.)

IPEC_RASReasonUndefined (0xfb3, 4019 decimal)
Unknown reason. (In GRJ, RRJ, URJ, ARJ, BRJ, LRJ, and INAK messages.)

IPEC_RASReasonTerminalExcluded (0xfb4, 4020 decimal)
Permission failure and not a resource failure. (In GRQ messages.)

IPEC_RASReasonNotBound (0xfb5, 4021 decimal)
Discovery permission has expired. (In BRJ messages.)

IPEC_RASReasonNotCurrentlyRegistered (0xfb6, 4022 decimal)
The endpoint is not registered. (In URJ messages.)

IPEC_RASReasonRequestDenied (0xfb7, 4023 decimal)
No bandwidth is available. (In ARJ messages.)

Unable to find location. (In LRJ messages.)

IPEC_RASReasonLocationNotFound (0xfb8, 4024 decimal)
Unable to find location. (In LRJ messages.)

IPEC_RASReasonSecurityDenial (0xfb9, 4025 decimal)
Security access has been denied. (In GRJ, RRJ, URJ, ARJ, BRJ, LRJ, DRJ, and INAK messages.)

IPEC_RASTransportQOSNotSupported (0xfba, 4026 decimal)
QOS is not supported by this gatekeeper. (In RRJ messages.)

IPEC_RASResourceUnavailable (0xfbb, 4027 decimal)
Resources have been exhausted. (In GRJ, RRJ, ARJ and LRJ messages.)

IPEC_RASInvalidAlias (0xfbc, 4028 decimal)
The alias is not consistent with gatekeeper rules. (In RRJ messages.)

IPEC_RASPermissionDenied (0xfbd, 4029 decimal)
The requesting user is not allowed to unregistered the specified user. (In URJ messages.)

IPEC_RASQOSControlNotSupported (0xfbe, 4030 decimal)
QOS control is not supported. (In ARJ messages.)

IPEC_RASIncompleteAddress (0xfbf, 4031 decimal)
The user address is incomplete. (In ARJ messages.)

IPEC_RASFullRegistrationRequired (0xfc0, 4032 decimal)
Registration permission has expired. (In RRJ messages.)

IPEC_RASRouteCallToSCN (0xfc1, 4033 decimal)
The call was routed to a switched circuit network. (In ARJ and LRJ messages.)

IPEC_RASAliasesInconsistent (0xfc2, 4034 decimal)
Multiple aliases in the request identify separate people. (In ARJ and LRJ messages.)

10.5 Failure Response Codes When Using SIP

The following failure response codes apply when using SIP. Each code is followed by a description. The codes are listed in code value order.

Request Failure Response Codes (4xx)

IPEC_SIPReasonStatus400BadRequest (0x1518, 5400 decimal)
SIP Request Failure Response 400 - Bad Request - The request could not be understood due to malformed syntax. The Reason-Phrase should identify the syntax problem in more detail, for example, “Missing Call-ID header field”.

IPEC_SIPReasonStatus401Unauthorized (0x1519, 5401 decimal)
SIP Request Failure Response 401 - Unauthorized - The request requires user authentication. This response is issued by User Agent Servers (UASs) and registrars, while 407 (Proxy Authentication Required) is used by proxy servers.

IPEC_SIPReasonStatus402PaymentRequired (0x151a, 5402 decimal)
SIP Request Failure Response 402 - Payment Required - Reserved for future use.

IPEC_SIPReasonStatus403Forbidden (0x151b, 5403 decimal)
SIP Request Failure Response 403 - Forbidden - The server understood the request, but is refusing to fulfill it. Authorization will not help, and the request should not be repeated.

IPEC_SIPReasonStatus404NotFound (0x151c, 5404 decimal)
SIP Request Failure Response 404 - Not Found - The server has definitive information that the user does not exist at the domain specified in the Request-URI. This status is also returned if the domain in the Request-URI does not match any of the domains handled by the recipient of the request.

IPEC_SIPReasonStatus405MethodNotAllowed (0x151d, 5405 decimal)
SIP Request Failure Response 405 - Method Not Allowed - The method specified in the Request-Line is understood, but not allowed for the address identified by the Request-URI. The response must include an Allow header field containing a list of valid methods for the indicated address.

IPEC_SIPReasonStatus406NotAcceptable (0x151e, 5406 decimal)
SIP Request Failure Response 406 - Not Acceptable - The resource identified by the request is only capable of generating response entities that have content characteristics not acceptable according to the Accept header field sent in the request.

IPEC_SIPReasonStatus407ProxyAuthenticationRequired (0x151f, 5407 decimal)
SIP Request Failure Response 407 - Proxy Authentication Required - This code is similar to 401 (Unauthorized), but indicates that the client must first authenticate itself with the proxy. This status code can be used for applications where access to the communication channel (for example, a telephony gateway) rather than the callee, requires authentication.

IPEC_SIPReasonStatus408RequestTimeout (0x1520, 5408 decimal)
SIP Request Failure Response 408 - Request Timeout - The server could not produce a response within a suitable amount of time, for example, if it could not determine the location of the user in time. The client may repeat the request without modifications at any later time.

IPEC_SIPReasonStatus410Gone (0x1522, 5410 decimal)
SIP Request Failure Response 410 - Gone - The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to be considered permanent. If the server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) should be used instead.

IPEC_SIPReasonStatus413RequestEntityTooLarge (0x1525, 5413 decimal)
SIP Request Failure Response 413 - Request Entity Too Large - The server is refusing to process a request because the request entity-body is larger than the server is willing or able to process. The server may close the connection to prevent the client from continuing the request. If the condition is temporary, the server should include a Retry-After header field to indicate that it is temporary and after what time the client may try again.

IPEC_SIPReasonStatus414RequestUriTooLong (0x1526, 5414 decimal)
SIP Request Failure Response 414 - Request-URI Too Long - The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.

IPEC_SIPReasonStatus415UnsupportedMediaType (0x1527, 5415 decimal)
SIP Request Failure Response 415 - Unsupported Media Type - The server is refusing to service the request because the message body of the request is in a format not supported by the server for the requested method. The server must return a list of acceptable formats using the Accept, Accept-Encoding, or Accept-Language header field, depending on the specific problem with the content.

IPEC_SIPReasonStatus416UnsupportedURIScheme (0x1528, 5416 decimal)
SIP Request Failure Response 416 - Unsupported URI Scheme - The server cannot process the request because the scheme of the URI in the Request-URI is unknown to the server.

IPEC_SIPReasonStatus420BadExtension (0x153c, 5420 decimal)
SIP Request Failure Response 420 - Bad Extension - The server did not understand the protocol extension specified in a Proxy-Require or Require header field. The server must include a list of the unsupported extensions in an Unsupported header field in the response.

IPEC_SIPReasonStatus421ExtensionRequired (0x153d, 5421 decimal)
SIP Request Failure Response 421 - Extension Required - The User Agent Server (UAS) needs a particular extension to process the request, but this extension is not listed in a Supported header field in the request. Responses with this status code must contain a Require header field listing the required extensions. A UAS should not use this response unless it truly cannot provide any useful service to the client. Instead, if a desirable extension is not listed in the Supported header field, servers should process the request using baseline SIP capabilities and any extensions supported by the client.

IPEC_SIPReasonStatus423IntervalTooBrief (0x153f, 5423 decimal)
SIP Request Failure Response 423 - Interval Too Brief - The server is rejecting the request because the expiration time of the resource refreshed by the request is too short. This response can be used by a registrar to reject a registration whose Contact header field expiration time was too small.

IPEC_SIPReasonStatus480TemporarilyUnavailable (0x1568, 5480 decimal)
SIP Request Failure Response 480 - Temporarily Unavailable - The callee's end system was contacted successfully but the callee is currently unavailable (for example, is not logged in, logged in but in a state that precludes communication with the callee, or has activated the “do not disturb” feature). The response may indicate a better time to call in the Retry-After header field. The user could also be available elsewhere (unbeknownst to this server). The reason

phrase should indicate a more precise cause as to why the callee is unavailable. This value should be settable by the User Agent (UA). Status 486 (Busy Here) may be used to more precisely indicate a particular reason for the call failure. This status is also returned by a redirect or proxy server that recognizes the user identified by the Request-URI, but does not currently have a valid forwarding location for that user.

IPEC_SIPReasonStatus481CallTransactionDoesNotExist (0x1569, 5481 decimal)
SIP Request Failure Response 481 - Call/Transaction Does Not Exist - This status indicates that the User Agent Server (UAS) received a request that does not match any existing dialog or transaction.

IPEC_SIPReasonStatus482LoopDetected (0x156a, 5482 decimal)
SIP Request Failure Response 482 - Loop Detected - The server has detected a loop.

IPEC_SIPReasonStatus483TooManyHops (0x156b, 5483 decimal)
SIP Request Failure Response 483 - Too Many Hops - The server received a request that contains a Max-Forwards header field with the value zero.

IPEC_SIPReasonStatus484AddressIncomplete (0x156c, 5484 decimal)
SIP Request Failure Response 484 - Address Incomplete - The server received a request with a Request-URI that was incomplete. Additional information should be provided in the reason phrase. This status code allows overlapped dialing. With overlapped dialing, the client does not know the length of the dialing string. It sends strings of increasing lengths, prompting the user for more input, until it no longer receives a 484 (Address Incomplete) status response.

IPEC_SIPReasonStatus485Ambiguous (0x156d, 5485 decimal)
SIP Request Failure Response 485 - The Request-URI was ambiguous. The response may contain a listing of possible unambiguous addresses in Contact header fields. Revealing alternatives can infringe on privacy of the user or the organization. It must be possible to configure a server to respond with status 404 (Not Found) or to suppress the listing of possible choices for ambiguous Request-URIs.

IPEC_SIPReasonStatus486BusyHere (0x156e, 5486 decimal)
SIP Request Failure Response 486 - Busy Here - The callee's end system was contacted successfully, but the callee is currently not willing or able to take additional calls at this end system. The response may indicate a better time to call in the Retry-After header field. The user could also be available elsewhere, such as through a voice mail service. Status 600 (Busy Everywhere) should be used if the client knows that no other end system will be able to accept this call.

IPEC_SIPReasonStatus487RequestTerminated (0x156f, 5487 decimal)
SIP Request Failure Response 487 - Request Terminated - The request was terminated by a BYE or CANCEL request. This response is never returned for a CANCEL request itself.

IPEC_SIPReasonStatus488NotAcceptableHere (0x1570, 5488 decimal)
SIP Request Failure Response 488 - Not Acceptable Here - The response has the same meaning as 606 (Not Acceptable), but only applies to the specific resource addressed by the Request-URI and the request may succeed elsewhere. A message body containing a description of media capabilities may be present in the response, which is formatted according to the Accept header field in the INVITE (or application/SDP if not present), the same as a message body in a 200 (OK) response to an OPTIONS request.

IPEC_SIPReasonStatus491RequestPending (0x1573, 5491 decimal)
SIP Request Failure Response 491 - Request Pending - The request was received by a User Agent Server (UAS) that had a pending request within the same dialog.

IPEC_SIPReasonStatus493Undecipherable (0x1575, 5493 decimal)

SIP Request Failure Response 493 - Undecipherable - The request was received by a User Agent Server (UAS) that contained an encrypted MIME body for which the recipient does not possess or will not provide an appropriate decryption key. This response may have a single body containing an appropriate public key that should be used to encrypt MIME bodies sent to this User Agent (UA).

Server Failure Response Codes (5xx)

IPEC_SIPReasonStatus500ServerInternalError (0x157c, 5500 decimal)

Server Failure Response 500 - Server Internal Error - The server encountered an unexpected condition that prevented it from fulfilling the request. The client may display the specific error condition and may retry the request after several seconds. If the condition is temporary, the server may indicate when the client may retry the request using the Retry-After header field.

IPEC_SIPReasonStatus501NotImplemented (0x157d, 5501 decimal)

Server Failure Response 501 - Not Implemented - The server does not support the functionality required to fulfill the request. This is the appropriate response when a User Agent Server (UAS) does not recognize the request method and is not capable of supporting it for any user. Proxies forward all requests regardless of method. Note that a 405 (Method Not Allowed) is sent when the server recognizes the request method, but that method is not allowed or supported.

IPEC_SIPReasonStatus502BadGateway (0x157e, 5502 decimal)

Server Failure Response 502 - Bad Gateway - The server, while acting as a gateway or proxy, received an invalid response from the downstream server it accessed in attempting to fulfill the request.

IPEC_SIPReasonStatus503ServiceUnavailable (0x157f, 5503 decimal)

Server Failure Response 503 - Service Unavailable - The server is temporarily unable to process the request due to a temporary overloading or maintenance of the server or the use of an unsupported transport protocol (for example, TCP). The server may indicate when the client should retry the request in a Retry-After header field. If no Retry-After is given, the client must act as if it had received a 500 (Server Internal Error) response. A client (proxy or User Agent Client) receiving a 503 (Service Unavailable) should attempt to forward the request to an alternate server. It should not forward any other requests to that server for the duration specified in the Retry-After header field, if present. Servers may refuse the connection or drop the request instead of responding with 503 (Service Unavailable).

IPEC_SIPReasonStatus504ServerTimeout (0x1580, 5504 decimal)

Server Failure Response 504 - Server Time-out - The server did not receive a timely response from an external server it accessed in attempting to process the request. 408 (Request Timeout) should be used instead if there was no response within the period specified in the Expires header field from the upstream server.

IPEC_SIPReasonStatus505VersionNotSupported (0x1581, 5505 decimal)

Server Failure Response 505 - Version Not Supported - The server does not support, or refuses to support, the SIP protocol version that was used in the request. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, other than with this error message.

IPEC_SIPReasonStatus513MessageTooLarge (0x1589, 5513 decimal)

Server Failure Response 513 - Message Too Large - The server was unable to process the request since the message length exceeded its capabilities.

Global Failure Response Codes (6xx)

IPEC_SIPReasonStatus600BusyEverywhere (0x15e0, 5600 decimal)

SIP Global Failure Response 600 - Busy Everywhere - The callee's end system was contacted successfully but the callee is busy and does not wish to take the call at this time. The response may indicate a better time to call in the Retry-After header field. If the callee does not wish to reveal the reason for declining the call, the callee uses status code 603 (Decline) instead. This status response is returned only if the client knows that no other end point (such as a voice mail system) will answer the request. Otherwise, 486 (Busy Here) should be returned.

IPEC_SIPReasonStatus603Decline (0x15e3, 5603 decimal)

SIP Global Failure Response 603 - 603 Decline - The callee's machine was successfully contacted but the user explicitly does not wish to or cannot participate. The response may indicate a better time to call in the Retry-After header field. This status response is returned only if the client knows that no other end point will answer the request.

IPEC_SIPReasonStatus604DoesNotExistAnywhere (0x15e4, 5604 decimal)

SIP Global Failure Response 604 - Does Not Exist Anywhere - The server has authoritative information that the user indicated in the Request-URI does not exist anywhere.

IPEC_SIPReasonStatus606NotAcceptable (0x15e6, 5606 decimal)

SIP Global Failure Response 606 - Not Acceptable - The user's agent was contacted successfully but some aspects of the session description such as the requested media, bandwidth, or addressing style were not acceptable. A 606 (Not Acceptable) response means that the user wishes to communicate, but cannot adequately support the session described.

The 606 (Not Acceptable) response may contain a list of reasons in a Warning header field describing why the session described cannot be supported.

A message body containing a description of media capabilities may be present in the response, which is formatted according to the Accept header field in the INVITE (or application/SDP if not present), the same as a message body in a 200 (OK) response to an OPTIONS request.

It is hoped that negotiation will not frequently be needed, and when a new user is being invited to join an already existing conference, negotiation may not be possible. It is up to the invitation initiator to decide whether or not to act on a 606 (Not Acceptable) response.

This status response is returned only if the client knows that no other end point will answer the request.

Other SIP Codes (8xx)

IPEC_SIPReasonStatusBYE (0x16a8, 5800 decimal)

SIP reason status 800. BYE code.

IPEC_SIPReasonStatusCANCEL (0x16a9, 5801 decimal)

SIP reason status 801. CANCEL code.

SIP MIME Error Codes

IPEC_MIME_BUFF_TOO_SMALL
MIME buffer size is smaller than the incoming MIME part in a SIP message.

IPEC_MIME_POOL_EMPTY
MIME memory pool is exhausted.

SIP Registration Error Codes

IPEC_REG_FAIL_insufficientInternalResources
The SIP stack ran out of resources to process request.

IPEC_REG_FAIL_internalError
An internal IP Call Control Library error was encountered while attempting to form an outgoing REGISTER request.

IPEC_REG_FAIL_invalidExpires
The value of the “expires=” parameter in the Contact: header field was invalid for the current operation.

IPEC_REG_FAIL_networkError
A network error prevented the REGISTER request from being sent.

IPEC_REG_FAIL_registrationTransactionInProgress
A REGISTER transaction is currently in progress with the specified Registrar and Address of Record. A new request to this same Registrar and AOR cannot be generated at this time, and you should try again after the current pending request completes.

IPEC_REG_FAIL_responseTimeout
There was a timeout error while waiting for a REGISTER response from the Registrar.

IPEC_REG_FAIL_serverResponseDataMismatch
There was a mismatch between the internal IP Call Control library data and the data contained in the Registrar’s response.

Supplementary Reference Information 11

This chapter lists related publications and includes other reference information as follows:

11.1 References to More Information

The following publications provide related information:

- ITU-T Recommendation H.323 (11/00) - Packet-based multimedia communications systems
- ITU-T Recommendation H.245 (07/01) - Control protocol for multimedia communication
- ITU-T Recommendation H.225.0 (09/99) - Call signaling protocols and media stream packetization for packet-based multimedia communications systems
- ITU-T Recommendation T.38 (06/98) - Procedures for real-time Group 3 facsimile communication over IP networks
- ITU-T Recommendation T.30 (07/96) - Procedures for document facsimile transmission in the general switched telephone network
- RFC 1889, *RTP: A Transport Protocol for Real-Time Applications*, IETF, *http://ietf.org/rfc/rfc1889.txt*
- RFC 2976, *The SIP INFO Method*, IETF, *http://ietf.org/rfc/rfc2976.txt*
- RFC 3261, *Session Initiation Protocol (SIP)*, IETF, *http://ietf.org/rfc/rfc3261.txt*
- RFC 3265, *Session Initiation Protocol (SIP)-Specific Event Notification* [SUBSCRIBE and NOTIFY methods], IETF, *http://ietf.org/rfc/rfc3265.txt*
- RFC 3515, *The Session Initiation Protocol (SIP) Refer Method*, IETF, *http://ietf.org/rfc/rfc3515.txt*
- Cisco Systems, *Signaled Digits in SIP*, draft reference *http://www.ietf.org/internet-drafts/draft-mahy-sipping-signaled-digits-00.txt*
- Black, Uyless, *Voice over IP*, Prentice Hall PTR, Prentice-Hall, Inc. (Copyright 2000)
- Douskalis, Bill, *IP Telephony; The Integration of Robust VoIP Services*, Prentice Hall PTR, Prentice-Hall, Inc., ISBN 0-13-014118-6
- Galtieri, Paolo*, Introduction to Voice Over the Internet Protocol*, Applied Computing Technologies, Winter 2000

11.2 Called and Calling Party Address List Format When Using H.323

This section provides reference information about called and calling party address list format:

- Called Party Address List
- Calling Party Address List
- Examples of Called and Calling Party Addresses

Called Party Address List

Called party address lists are formatted as follows:

```
Called Party Address list ::= Called Party Address |
    Called Party Address Delimiter Party Address list

Called Party Address ::= Dialable Address | Name |
    E164ALIAS | Extension | Subaddress | Transport
    Address | Email Address | URL | Party Number |
    Transport Name
```

where:

- Dialable Address ::= E164Address | E164Address ";" Dialable Address
- Name ::= "NAME:" H323ID
- E164ALIAS ::= "TEL:" E164Address
- Extension ::= "EXT:" E164Address | "EXTID : " H323ID
- Subaddress ::= "SUB:" E164Address
- Transport Address ::= "TA:" Transport Address Spec | "FTH : " Transport address Spec.
  - Transport Address Spec ::= Host Name":" Port Number | Host Name
    - Host Name ::= Host IP in decimal dotted notation.
- Email Address ::= "EMAIL :" email address
- URL Address ::= "URL : " URL
- PN Address ::= "PN :" party number ["$" party number type]

- Party Number Type ::= (select either the numerical or string value from the following list):
    - **0.PUU** - The numbering plan follows the E.163 and E.164Recommendations.
    - **PUI** - The number digits carry a prefix indicating type of number according to national recommendations.
    - **PUN** - The number digits carry a prefix indicating the type of number according to national recommendations.
    - **PUNS** - The number digits carry a prefix indicating the type of number according to network specifications.
    - **PUA** - Valid only for the called party number at the outgoing access; the network substitutes appropriate number.
    - **D** - Valid only for the called party number at the outgoing access; the network substitutes appropriate number.
    - **PRL2** - Level 2 regional subtype of private number.
    - **PRL1** - Level 1 regional subtype of private number.
    - **PRP** - PISN subtype of private number.
    - **PRL** - Local subtype of private number.
    - **PRA** - Abbreviated subtype of private number.
    - **N** - The number digits carry a prefix indicating standard type of number according to national recommendations.
- Transport Name ::= “TNAME :” Transport Address Spec

Notes: ***1.*** The delimiter is “,” by default, but it may be changed by setting the value of the delimiter field in the IPCCLIB_START_DATA used by the **gc_Start( )** function. See Section 7.3.26, “gc_Start( ) Variances for IP”, on page 340 for more information.

2. If the Dialable Address form of the address is used, it should be the last item in the list of address alternatives.

Calling Party Address List

Calling party address lists are formatted as follows:

```
Calling Party address list ::= Calling Party address |
    Calling Party address Delimiter |
    Calling Party address list

Calling Party address ::= Dialable Address | Name |
    E164ALIAS | Extension | Subaddress | Transport
    Address | Email Address | URL | Party Number |
    Transport Name
```

where the format options Dialable Address, Name, etc. are as described in the Called Party Address List section.

Note: If the Dialable Address form of the Party address is used, it should be the last item in the list of Party address alternatives.

Examples of Called and Calling Party Addresses

Some examples of called party and calling party addresses are:

- Called and Calling Party addresses: 1111;1111
- NAME: John, NAME: Jo
- TA:192.114.36.10

Glossary

alias: A nickname for a domain or host computer on the Internet.

blind transfer: See *unsupervised transfer.*

call transfer: See *supervised transfer* and *unsupervised transfer.*

codec: A device that converts analog voice signals to a digital form and vice versa. In this context, analog signals are converted into the payload of UDP packets for transmission over the internet. The codec also performs compression and decompression on a voice stream.

H.225.0: Specifies messages for call control including signaling, Registration Admission and Status (RAS), and the packetization and synchronization of media streams.

en-bloc mode: A mode where the setup message contains all the information required by the network to process the call, such as the called party address information.

H.245: H.245 is a standard that provides the call control mechanism that allows H.323-compatible terminals to connect to each other. H.245 provides a standard means for establishing audio and video connections. It specifies the signaling, flow control, and channeling for messages, requests, and commands. H.245 enables codec selection and capability negotiation within H.323. Bit rate, frame rate, picture format, and algorithm choices are some of the elements negotiated by H.245.

gateway: Translates communication procedures and formats between networks, for example the interface between an IP network and the circuit-switched network (PSTN).

Gatekeeper: Manages a collection of H.323 entities (terminals, gateway, multipoint control units) in an H.323 zone.

H.255.0: The H.255.0 standard defines a layer that formats the transmitted audio, video, data, and control streams for output to the network, and retrieves the corresponding streams from the network.

H.323: H.323 is an ITU recommendation for a standard for interoperability in audio, video and data transmissions as well as Internet phone and voice-over-IP (VoIP). H.323 addresses call control and management for both point-to-point and multipoint conferences as well as gateway administration of IP Media traffic, bandwidth and user participation.

IP: Internet Protocol

IP Media Library: Intel API library used to control RTP streams.

Multipoint Control Unit (MCU): An endpoint that support conferences between three or more endpoints.

prefix: One or several digits dialed in front of a phone number, usually to indicate something to the phone system. For example, dialing a zero in front of a long distance number in the United States indicates to the phone company that you want operator assistance on a call.

Q.931: The Q.931 protocol defines how each H.323 layer interacts with peer layers, so that participants can interoperate with agreed upon formats. The Q.931 protocol resides within H.225.0. As part of H.323 call control, Q.931 is a link layer protocol for establishing connections and framing data.

RTP: Real-time Transport Protocol. Provides end-to-end network transport functions suitable for applications transmitting real-time data such as audio, video or simulation data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services.

RTCP: RTP Control Protocol (RTCP). Works in conjunction with RTP to allow the monitoring of data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets.

silence suppression: See Voice Activation Detection (VAD).

supervised transfer: A call transfer in which the person transferring the call stays on the line, announces the call, and consults with the party to whom the call is being transferred before the transfer is completed.

UA: In a SIP context, user agents (UAs) are appliances or applications, such as, SIP phones, residential gateways and software that initiate and receive calls over a SIP network.

SIP: Session Initiated Protocol. An ASCII-based, peer-to-peer protocol designed to provide telephony services over the Internet.

split call control: An IP telephony software architecture in which call control is done separately from IP Media stream control, for example, call control is done on the host and IP Media stream control is done on the board.

tunneling: The encapsulation of H.245 messages within Q.931/H.225 messages so that H.245 media control messages can be transmitted over the same TCP port as the Q.931/H.225 signaling messages.

unsupervised transfer: A transfer in which the call is transferred without any consultation or announcement by the person transferring the call.

VAD: Voice Activation Detection. In Voice over IP (VoIP), voice activation detection (VAD) is a technique that allows a data network carrying voice traffic over the Internet to detect the absence of audio and conserve bandwidth by preventing the transmission of *silent packets* over the network.

Index

New Feature Documentation Locations, IP Technology Guide for HMP 1.1 SU

SIP Call Transfer

Section 3.3, “Call Transfer Scenarios When Using SIP”

Section 4.23.5, “Call Transfer When Using SIP”

SIP Outbound Proxy

Section 4.1.1, “Setting a SIP Outbound Proxy”

plus

- IP_VIRTBOARD structure description

SIP over TCP

Section 4.1.2, “Configuring SIP Transport Protocol”

Section 4.8, “Specifying Transport for SIP Messages”

plus

- IP_VIRTBOARD structure description

SIP Request Retry

Section 4.9, “Handling SIP Transport Failures”

plus

- IP_VIRTBOARD structure description

Access to Additional SIP Message Headers

Section 4.6, “Setting and Retrieving SIP Message Headers”

plus

- Section 8.1, “Overview of Parameter Usage”
- Section 8.2.22, “IPSET_SIP_MSGINFO”

MIME-Encoded SIP Message Bodies (SIP-T)

Section 4.7, “Using SIP Messages with MIME Bodies (SIP-T)”

plus

- Section 8.1, “Overview of Parameter Usage”
- Section 8.2.12, “IPSET_MIME and IPSET_MIME_200OK_TO_BYE”

SIP INFO Messages

Section 4.10, “Sending and Receiving SIP INFO Messages”

plus

- Section 4.6, “Setting and Retrieving SIP Message Headers”
- Section 4.7, “Using SIP Messages with MIME Bodies (SIP-T)”
- Section 8.2.16, “IPSET_MSG_SIP”

SIP OPTIONS Messages

Section 4.11, “Sending and Receiving SIP OPTIONS Messages”

plus

- Section 4.6, “Setting and Retrieving SIP Message Headers”
- Section 4.7, “Using SIP Messages with MIME Bodies (SIP-T)”
- Section 8.2.16, “IPSET_MSG_SIP”

SIP SUBSCRIBE and NOTIFY Messages

Section 4.12, “Using SIP SUBSCRIBE and NOTIFY Messages”

plus

- Section 4.6, “Setting and Retrieving SIP Message Headers”

- Section 4.7, “Using SIP Messages with MIME Bodies (SIP-T)”
- Section 8.2.16, “IPSET_MSG_SIP”

Getting RTP Addresses of a Call
Section 4.14, “Getting Media Streaming Status and Connection Information”

plus

- Section 8.2.21, “IPSET_RTP_ADDRESS”

Getting SIP-specific Origination and Destination Addresses for a Call
Section 7.2.9, “gc_GetCallInfo( ) Variances for IP”

Host LAN Cable Disconnect Alarm
Section 4.26, “Host LAN Disconnection Alarms”

RTF Logging
Chapter 6, “Debugging Global Call IP Applications”